Software Architecture Evolution

Jeffrey M. Barnes

December 2013

CMU-ISR-13-118

Institute for Software Research
School of Computer Science
Carnegie Mellon University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This dissertation is based on research supported by the Software Engineering Institute; the Jet Propulsion Laboratory Summer Internship Program; the National Aeronautics and Space Administration; IBM; the National Science Foundation under grant 0615305; and the United States Navy under contracts N000141310401, N000141310171, and N000140811223. The views and conclusions contained herein are my own and should not be interpreted as representing the policies or opinions of Carnegie Mellon University or any of the sponsors of this research.

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.		
1. REPORT DATE **DEC 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**
4. TITLE AND SUBTITLE **Software Architecture Evolution**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Institute for Software Research,Pittsurgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **Many software systems eventually undergo changes to their basic architectural structure. Such changes may be prompted by new feature requests, new quality attribute requirements, changing technology, or other reasons. Whatever the causes, architecture evolution is commonplace in real-world software projects. Today?s software architects, however, have few techniques to help them plan such evolution. In particular, they have little assistance in planning alternatives, making trade-offs among these different alternatives, or applying best practices for particular domains. To address this, we have developed an approach for assisting architects in planning and reasoning about software architecture evolution. Our approach is based on modeling and analyzing potential evolution paths that represent different ways of evolving the system. We represent an evolution path as a sequence of transitional architectural states leading from the initial architecture to the target architecture along with evolution operators that characterize the transitions among these states. We support analysis of evolution paths through the definition and application of constraints that express rules governing the evolution of the systemand evaluation functions that assess path quality. Finally, a set of these modeling elements may be grouped together into an evolution style that encapsulates a body of knowledge relevant to a particular domain of architecture evolution. We evaluate this approach in three ways. First, we evaluate its applicability to real-world architecture evolution projects. This is accomplished through case studies of two very different software organizations. Second, we undertake a formal evaluation of the computational complexity of verifying evolution constraints. Finally, we evaluate the implementability of the approach based on our experiences developing prototype tools for software architecture evolution.**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**238**	

Keywords: software architecture, software evolution, software engineering, formal methods, specification, temporal logic, case study, content analysis

Abstract

Many software systems eventually undergo changes to their basic architectural structure. Such changes may be prompted by new feature requests, new quality attribute requirements, changing technology, or other reasons. Whatever the causes, architecture evolution is commonplace in real-world software projects.

Today's software architects, however, have few techniques to help them plan such evolution. In particular, they have little assistance in planning alternatives, making trade-offs among these different alternatives, or applying best practices for particular domains.

To address this, we have developed an approach for assisting architects in planning and reasoning about software architecture evolution. Our approach is based on modeling and analyzing potential *evolution paths* that represent different ways of evolving the system. We represent an evolution path as a sequence of transitional architectural states leading from the initial architecture to the target architecture, along with *evolution operators* that characterize the transitions among these states. We support analysis of evolution paths through the definition and application of *constraints* that express rules governing the evolution of the system and *evaluation functions* that assess path quality. Finally, a set of these modeling elements may be grouped together into an *evolution style* that encapsulates a body of knowledge relevant to a particular domain of architecture evolution.

We evaluate this approach in three ways. First, we evaluate its *applicability* to real-world architecture evolution projects. This is accomplished through case studies of two very different software organizations. Second, we undertake a formal evaluation of the *computational complexity* of verifying evolution constraints. Finally, we evaluate the *implementability* of the approach based on our experiences developing prototype tools for software architecture evolution.

This dissertation is dedicated to the memory of my grandfather.

Lester Earl Barnes, Jr.

June 20, 1929 – November 24, 2013

When I was in fifth grade, Grandpa Barnes gave me a book on computer programming and turned me loose on the QBasic interpreter that came packaged with MS-DOS in the 1990s, setting me on the very long journey that has led me to a PhD in software engineering. He has been a source of wisdom and encouragement throughout my life and will be dearly missed.

Acknowledgments

Appropriately enough, I am writing these acknowledgments on Thanksgiving Day, four days before my thesis defense. This Thanksgiving, I have much to be grateful for.

I would like to begin by thanking my thesis advisor, David Garlan, who has been my principal resource and guide throughout my graduate studies. I could not have hoped for a better advisor. Prospective students visiting Carnegie Mellon often ask about David's advising style. Is he hands-on or hands-off? Big-picture or detail-oriented? I'm always at a bit of a loss about how to answer such questions. The truth is that David is a very versatile advisor. He has a remarkable knack for providing the right kind of help and the right amount of guidance at the right time. He's a fantastic source of grand visions and big ideas, but he is also abundantly heedful of the need to get the small details right. He'll help take you step-by-step through an unfamiliar process when you need it, but he'll also entrust you with freedom and responsibility once you've learned the ropes. He's visionary and idealistic when vision and idealism are what's needed, and pragmatic and hardheaded when the time comes for action. I am deeply grateful for the guidance he has given me over the last six years.

The other members of my thesis committee have done much to improve the quality of this work. Travis Breaux was particularly helpful in providing guidance on research methodology and in encouraging me to carry out my empirical evaluation as meticulously and thoughtfully as I could.

Ipek Ozkaya has been supportive of my research and influential in directing its course since long before I asked her to join my thesis committee. She has been a continual source of helpful advice, guidance, and encouragement throughout my graduate studies.

Finally, my external committee member, Kevin Sullivan, provided a number of useful comments that have helped to shape the direction of my thesis research. I am grateful to all four of my thesis committee members for their insight and guidance.

Over the years, I have been fortunate to receive useful input on my research from a great many people. I would like to particularly acknowledge the contributions of Bradley Schmerl, Orieta Celiku, Brian Giovannoni, Dave Santo, Oleg Sindiy, S Sivakumar, Sridhar Iyengar, and Shrikant Palkar. In addition, individuals who contributed significantly to specific projects are acknowledged in footnotes within the relevant chapters of this dissertation.

I would also like to thank past and present members of the ABLE research group, as well as the students, faculty, and staff of the Institute for Software Research at large. I very much appreciate the supportive, collegial environment that ISR has provided for me during my time here.

The title page of this dissertation identifies the funding sources that made this research possible. I am very grateful to all the organizations that have provided support

for this work. I would like to particularly express my gratitude to the Software Engineering Institute. The consistent support provided by the SEI for this research over the last several years has enabled me to examine the topic of software architecture evolution in the depth and breadth that it deserves.

Finally, I will be forever grateful for the emotional, financial, and practical support that my family has provided me throughout my education. Mom, Dad, and Kevin: thank you! Your love and support have meant more to me than you know.

Most of all, I am thankful for Mina, who has been a constant source of inspiration, a beacon of stability, and the light at the end of my tunnel. It's been a long, crazy journey together, and I'm grateful every day to have you at my side.

Contents

1 Introduction

Architectural change is commonplace in real-world software systems. However, today's software architects have few tools to help them to plan such architecture evolution effectively. While considerable research has gone into software maintenance and evolution generally, there has been relatively little work focusing specifically on foundations and tools to support architecture evolution.

In particular, architects planning a major evolution have almost no assistance in reasoning about questions such as: How should we stage the evolution to achieve business goals in the presence of limited development resources? How can we ensure that intermediate releases do not break existing functionality? How can we reduce risk in incorporating new technologies and infrastructure required by the target architecture? How can we make principled trade-offs between time and development effort? What kinds of changes can be made independently, and which require coordinated systemwide modifications? How can we represent and communicate an evolution plan within an organization?

We have developed an approach to support architects in planning and reasoning about software evolution. Our approach is based on modeling and analyzing candidate *evolution paths*—plans for evolution leading from the current state of the system to a desired target state. The rest of this chapter elaborates on the motivation behind this work. Section 1.1 elaborates on the problem that this work aims to address. Section 1.2 presents an illustrative example to explain the basic elements of the proposed approach. Section 1.3 states the thesis of this dissertation. Finally, section 1.4 describes the structure of the remainder of this document, explaining each chapter's relevance to the stated thesis.

1.1 Problem

Rearchitecting a software system is often a complex, multifaceted operation; typically, an architect preparing to carry out an evolution must develop a plan to change the architecture and implementation of a system through a series of phased releases, ultimately leading to the target architecture. The architect's task, therefore, is to develop a plan consisting of a series of architectural designs for intermediate states of the system as it evolves, beginning with the current state of the system and ending with the target state, together with a characterization of the engineering operations required to transform the system from its current architecture to the target architecture. This idea is the basis for the model of software architecture evolution that underlies this research.

It is assumed that both the initial architecture (the current design of the system) and the target architecture (the intended design to which the system must evolve) are known. In fact, of course, this is often not the case. Especially in the case of an old, legacy system, design documents showing the initial architecture may be lost, fragmentary, or outdated, or they may never have existed at all. And similarly, the architect may be uncertain what the target architecture should be. However, other areas of research address these problems. The problem of learning the architecture of an existing system for which no architectural representation currently exists is addressed by research on *architecture reconstruction*, which provides tools for extracting an architectural representation of an implementation [75]. And the problem of determining a target architecture is addressed by research on *architectural design*, which provides guidance on designing a software architecture that meets a set of requirements [21]. The research described in this dissertation instead explores the problem of finding a way to evolve the system from a known initial architecture to a chosen target architecture—of charting a course from a known origin to a known destination.

More specifically, this work addresses the following research questions:

RQ1. How can we model the various evolution paths available to a software architect who is planning a large-scale evolution of a system?

RQ2. How can we model and enforce constraints about which evolution paths are appropriate or permissible?

RQ3. How can we facilitate analysis of evolution path quality in a way that supports architects in making trade-offs among candidate evolution paths and selecting the optimal path?

RQ4. How can we capture expertise about domains of software architecture evolution so that it can be later reused to inform future evolution planning?

In this dissertation, I describe an approach for software architecture evolution that addresses these questions. The approach is based on a model in which evolution plans are explicitly represented as paths through a graph of potential intermediate states of evolution. The edges of this graph are defined in terms of operators that characterize the architectural transformations that form the evolution paths. In addition, constraints may be defined that specify which evolution paths are legal, and analyses may be defined to assist with selection of an optimal path. These operators, constraints, and analyses may be bundled together in reusable evolution styles that capture architectural expertise about domains of software evolution.

1.2 Motivating example

To illustrate the basic elements of the approach that this dissertation will present, consider the following fictitious, but representative, scenario: Company C runs an algorithmic trading platform with an aging software architecture. Its clients, mostly fund managers, use the platform to research, develop, and execute high-frequency trading algorithms. (In high-frequency trading, trading algorithms are programs, based on proprietary trading strategies, that automatically make trades in response to market conditions.) Currently, these various features are accessed via separate web interfaces, which are showing their age. Input of trading algorithms to be executed is accomplished via one interface, retrieval and analysis of market data through another. A third interface allows clients to download a desktop analysis toolkit that they can use to backtest candidate trading algorithms (i.e., use collected market data to test how they would have performed in the past). The interfaces are separated in this way for historical reasons, and while the separation makes maintenance easy (because the features can be maintained independently), clients hate it; they would rather be able to research market history, backtest possible trading strategies, define algorithms, and activate algorithms for execution on one site.

The current architecture has other problems too. First, while software maintenance is easy, maintaining the hardware is quite expensive. Indeed, many of the components of the system are hardware-intensive. Running the trading algorithms, for example, is quite processor-intensive, while storing the company's vast archive of market data requires a great deal of disk space. The hardware requires frequent upgrades; the company must have top-of-the-line computing hardware to keep up with its competitors. In addition, there are significant demand spikes, and the hardware the company has cannot always keep up. Recently, a hardware failure brought down one of the web dashboards for two full business days, enraging clients.

In order to address these concerns, Company C is considering migrating to a cloud-computing-based architecture. In the cloud-computing model, computing resources are sold by third parties as services and accessed over the Internet. For example, rather than maintaining its own infrastructure, a company can pay a cloud service to provide the infrastructure for them. Concretely, what this means is that a cloud platform like Amazon Web Services will host Company C's software systems on its own infrastructure, providing whatever resources Company C needs and is willing to pay for: data storage, computing capacity, bandwidth, and so on. But unlike a traditional hosting environment, cloud-computing resources are sold on demand (e.g., computing capacity is sold by the hour; storage is sold by the gigabyte-month) and are provisioned elastically (so a customer can have as much or as little of a service as needed).

By making use of such a platform, Company C could host its software in the cloud rather than maintaining infrastructure in-house—effectively outsourcing hardware maintenance while maintaining complete control of the software. Such a migration could solve many of the company's problems. Reliability would be assured by the cloud provider's service-level agreements. Upgrades to better hardware could be effected immediately on request. Resources could be increased on demand as required by demand spikes. Specialized hardware suitable for specific applications, such as high-CPU hardware for trading-algorithm execution or high-memory hardware for backtesting, is easy to provision. If the company's needs change in the future, its infrastructure can change with them; the company is not locked into the infrastructure that it owns. Finally, Company C could focus on its business of developing a great trading platform rather than on the day-to-day problems of managing infrastructure. Also as part of the migration, the company plans to merge its multiple separate user interfaces to create one unified client experience.

The particulars of this example are contrived, but the scenario is a common one.

Cloud computing is a hot topic in the electronic-trading community, and trading platforms are increasingly asking how they can use cloud architectures to improve reliability, increase scalability, and allow themselves to focus on their core expertise rather than the business of keeping hardware running [128]. Moreover, although this example was framed in terms of an algorithmic trading system, largely the same concerns apply to a much broader category of systems. A great many organizations are migrating to the cloud, or contemplating migrations to the cloud, to address these same concerns.

The chief architect at C must develop a plan to carry out the evolution in a set of staged releases. Let us see how this might be accomplished using the approach that I mentioned in section 1.1 and that I will describe in detail in chapter 2.

In this approach, the architect would be supported by an *evolution style*, which encapsulates information about a domain of software architecture evolution. In this case, the evolution style would capture specialized information about the problem of migrating in-house ad hoc web applications to cloud-computing environments. Capitalizing on past experience in this area, the evolution style would identify the essential characteristics of the initial and target architectures. It would also identify a set of structure- and behavior-changing operations. Examples include the migration of data from an in-house database to a cloud data store, introduction of adapters as necessary to allow legacy subsystems to exist in the cloud, and reprovisioning of hardware within the hub as the company ramps up the nascent, cloud-based system to full production capacity. Finally, the style would specify a set of path constraints. These would capture the correctness conditions for a valid evolution path. Specifically they would express things like: in every release all existing functionality must continue to be available; data should be migrated before applications; all the old componentry should continue to exist for at least a week after the new cloud environment becomes accessible to users, so that fallback to the old system is possible.

How would the chief architect at C use the evolution style? The first step would likely be the definition of the initial and target architectures. Existing tools make it relatively easy to specify these architectures using standard architecture modeling and visualization techniques. At this point the evolution tools would check that these two architectures satisfy the pre- and postconditions required by the style, perhaps noting situations in which the target architecture is missing certain required structures or is otherwise malformed with respect to the target family.

Next, the architect starts filling in intermediate stages. By applying in sequence the operations defined by the evolution style, the architect defines the first intermediate release; in this case, that release might involve the construction of a skeletal cloud application alongside the legacy application. The tools would check that the release is well formed and that the path satisfies the constraints of the style, warning the architect when it identifies divergences. This process repeats until the architect has fully specified a set of releases and transitions to arrive at the target architecture. To define alternative paths, the architect could follow this same process repeatedly, or instead begin by defining potential intermediate states and then later draw the paths between them.

The architect also needs to make decisions about various trade-offs—for example, reconciling available resources (e.g., programmers) with the effort and time needed to create each release. To do this the architect uses one of several parameterized analyses provided by the evolution style. The analyses require the architect to select dimensions of concern and provide weighted utilities. With these annotations in hand, the tools calculate costs and utility, allowing the architect to explore alternative scenarios. Over time, as the evolution proceeds, the architect will update the values and perform recalculations, perhaps leading to revisions of the remaining releases on the path.

The architect also needs to make decisions about various trade-offs. For example:

- One possible alternative for evolving the system may be inexpensive but take a long time to complete, while another may be faster but more costly. The architect may want support for understanding this trade-off and choosing an option based on the company's priorities.
- It may be possible to take some shortcuts to finish the evolution faster, at the risk of possibly causing occasional service outages. Again, the architect may want to explore this trade-off between evolution duration on the one hand, and availability or risk on the other.
- It may be necessary to reconcile available resources (e.g., programmers) with the effort and time needed to create each release, to ensure that engineering resources are not overcommitted during the evolution.

To reason about such trade-offs, the architect can make use of parameterized evaluation functions provided by the evolution style. The evaluation functions require the architect to select dimensions of concern and provide weighted utilities. With these annotations in hand, the tools calculate costs and utility, allowing the architect to explore alternative scenarios. Over time, as the evolution proceeds, the architect will update the values and perform recalculations, perhaps leading to revisions of the remaining releases on the path.

1.3 Thesis

Section 1.3.1 states the thesis that this work aims to support. Section 1.3.2 breaks this thesis statement down and explains the research claims and requirements implied by each part.

1.3.1 Thesis statement

We can help software architects to plan large-scale evolutions of software systems by providing: (a) a theory to support architectural reasoning about software evolution, (b) languages and models for expressing evolution constraints and analyses, and (c) tools that allow easy application of this theoretical framework.

1.3.2 Explanation

In this section, I break this thesis statement down and explain each of its constituent parts.

We can help software architects to plan
large-scale evolutions of software systems . . .

Planning a major evolution of a software system presents significant challenges. Understanding the complexities of a large system composed of many software elements can be a daunting task. To plan the evolution of such a system, an architect must consider a dizzying array of interacting concerns, such as:

- The dependencies that exist among the elements of the system, and how to manage these dependencies as the system evolves
- How the system integrates with other software systems, and how to avoid integration issues while carrying out the evolution of the system
- How to order and stage the operations necessary to carry out the evolution
- Various kinds of constraints on the evolution, including technical constraints (such as constraints on the structure of the system or constraints on the order in which evolution operations may be carried out) as well as business constraints (such as cost and time)
- The need to preserve (or enhance) the system's functionality and quality attributes during the course of the evolution

Keeping all these considerations in mind to develop a plan for evolving the system can be extraordinarily difficult in a complex evolution project. Moreover, an architect typically has a multitude of choices available—different ways of evolving the system. The task of understanding the trade-offs among the various alternatives and selecting which alternative is optimal with respect to the goals of the particular evolution presents its own significant challenges.

This thesis proposes an approach to help the architect navigate this complex maze. In this dissertation, I argue that many of these difficulties can be ameliorated by providing architects with tools that can leverage collected information about particular domains of evolution and assist the architect in understanding the issues of an evolution, considering alternative ways of evolving the system from its initial architecture to a desired target architecture, and making trade-offs among these alternatives.

A foundational principle of our approach is that it is an *architecture-level*, rather than a code-level, solution. Planning the large-scale evolution of an industrial-scale software system is a high-level engineering task that requires an understanding of overall system structure, consideration of constraints on system design, and principled trade-offs among various qualities of concern. For all but the smallest applications, code-level approaches alone are inadequate for dealing with such issues (see section 7.1); it is necessary to reason about the system at an architectural level.

On the other hand, approaches based on project planning (see section 7.2) are too high-level to address the problems that we aim to address. While project-planning approaches can be useful for estimating the overall effort involved in carrying out a large-scale software evolution and for making high-level business trade-offs, they are not useful for making detailed architectural decisions about the structure of the evolving system or the order of architectural operations, nor are they useful for analyzing the effect of constraints on the evolving architecture.

Thus, to support detailed planning of large-scale evolutions of software systems, this thesis argues that it is necessary to take an architectural approach. The rest of the thesis statement explains the elements of this approach.

. . . by providing: (a) a theory to support
architectural reasoning about software evolution, . . .

The first element of the approach that this thesis describes is a theory—a formally defined framework for modeling instances of software architecture evolution.[1] One of the most basic concepts in software architecture evolution is an *evolution path*, a plan for evolving a software system from its current architecture to the target architecture. The software architect's task is to select the optimal evolution path from among the set of all possible evolution paths. (The meaning of *optimal* depends on the goals of the particular evolution at hand, but in general terms the aim is to maximize the *utility* of the evolution for some appropriate model of utility.) A theory of software architecture evolution, then, must provide a means of defining such evolution paths. Further, it must provide:

- A means of describing an evolution path in terms of the intermediate states through which the system will pass as the evolution is carried out

[1]The word *theory* carries some baggage, so it may be helpful to explain in more-formal terms what I mean by it. There is not complete agreement on what exactly constitutes a theory, but there is fairly broad consensus that a theory at least includes a set of constructs or entities, propositions that define how the constructs relate, explanations for the propositions, and a characterization of the scope of the theory (the domain to which it is applicable).

There are many different kinds of theories. A typology is presented by Gregor [104]. Gregor writes about the meaning of theory in the domain of information systems, but her work has been influential in software engineering as well [76; 206]. Gregor distinguishes among five types of theory: analytical (or descriptive) theories, explanatory theories, predictive theories, theories that are both explanatory and predictive, and prescriptive theories. The sort of theory I present in this dissertation is probably best characterized as a *prescriptive* theory. Gregor [104, p. 619] explains a prescriptive theory as a special kind of predictive theory for design and action:

> A special case of prediction exists where the theory provides a description of the method or structure or both for the construction of an artifact (akin to a recipe). The provision of the recipe implies that the recipe, if acted upon, will cause an artifact of a certain type to come into being.

The appropriateness of including prescription under the label *theory* has been debated. For example, March & Smith [145] prefer to reserve the term *theory* for explanatory and predictive theories, citing the usage of the term in the natural sciences; they distinguish theories from constructs, models, methods, and implementations. For further discussion, see Gregor [104] as well as Sjøberg et al. [206].

- A means of characterizing the architectural transitions that must be carried out in order to traverse an evolution path
- A means of constraining and evaluating evolution paths

Section 2 will discuss the approach more specifically. In particular, it will show how the requirements listed above can be met by viewing an architecture evolution scenario as a directed graph in which intermediate evolution states are represented as nodes, architectural transitions are represented as edges, and evolution paths appear as paths from the node representing the initial state to that representing the target state.

This theory provides a formal basis for thinking about software architecture evolution. By itself it is just a conceptual model, but it provides a foundation that will support the languages, models, and tools necessary to help software architects plan evolutions.

. . . (b) languages and models for expressing
evolution constraints and analyses, . . .

The second element of the approach is a set of languages and models for capturing constraints and analyses over evolution paths. Constraints arise naturally in the course of software architecture evolution; understanding and capturing these constraints, and modeling their effects on the evolution, is critical for making good decisions. Therefore, we need a language that is sufficiently expressive to capture the constraints that arise naturally in real-world evolution planning. At the same time, verifying evolution constraints expressed in the language must be computationally tractable, and in addition the language should not be too hard to learn (by comparison with other possible constraint languages, such as common temporal logics).

Similarly, the ability to specify analyses or evaluations of candidate evolution paths is necessary for making intelligent trade-offs. This requires a means of expressing analyses in a way that is sufficiently general to reuse analyses across many evolutions and sufficient to express analyses useful for evolution planning.

. . . and (c) tools that allow easy application of this theoretical framework.

The approach as I have described it so far would be sufficient to model and analyze architecture evolutions in principle, but it would not be useful in practical contexts without adequate tooling. Therefore, it is critical that the languages and models developed in this research be amenable to automated analysis. More precisely, they must be reasonably easy to automate in such a way that evaluation of typical constraints and analyses can be completed within reasonable time, even for industrial-scale software systems.

1.3.3 Research claims

The thesis statement defines the shape of the approach that this research takes, but it does not explicitly articulate specific claims about the usefulness and practicality of

	Research claim	Evaluation	Reference
1	Applicability	Case studies	Ch. 4–5
	The approach is applicable to the concerns that arise in real-world architecture evolution projects.		
2	Computational complexity	Formal proofs	Ch. 3
	The approach admits efficient verification of evolution constraints.		
3	Implementability	Prototype development	Ch. 6
	The approach lends itself to tool support.		

Table 1. *A summary of the research claims described in section 1.3.3, along with the methods by which they are evaluated and cross-references to the relevant chapters of the dissertation.*

that approach. That is, it does not define criteria for evaluating the approach. In this section, I define three specific research claims about the usefulness and practicality of the approach and explain how they will be evaluated. These are summarized in table 1.

Research claim 1 (applicability). *The approach is applicable to the concerns that arise in real-world architecture evolution projects.*

The approach proposed in this work defines a set of concepts for reasoning about architecture evolution as well as a set of specification languages for modeling these concepts. But it is not immediately clear that these concepts and specification languages are actually useful for capturing the concerns that arise in the real world. It might be that the kinds of concerns we had in mind when we developed this approach are quite dissimilar from those that arise in actual architecture evolution projects, and that our approach is ill suited to representing the evolution concerns that arise in practice.

To evaluate the applicability of our approach for modeling the concerns that arise in real architecture evolution projects, I conducted two case studies at large software organizations, in which I gathered data on architecture evolution projects at the organizations and then attempted to construct evolution models using our approach. These case studies are described in chapters 4 and 5.

Research claim 2 (computational complexity). *The approach admits efficient verification of evolution constraints.*

It isn't sufficient that architecture evolution scenarios be specifiable using our approach; it also has to be practical to reason about them. Thus, part of ensuring the practicality of our approach is evaluating the computational complexity of carrying out the necessary analysis of the evolution graph.

In chapter 3, I provide a detailed theoretical treatment of the language for specifying evolution path constraints, and I prove some results about the time complexity and space complexity of verifying evolution path constraints. I focus on constraints in particular because the question of the tractability of evaluation functions is a less interesting one, as I will explain at the beginning of chapter 3.

Research claim 3 (implementability)**.** *The approach lends itself to tool support.*

Another aspect of practicality is ensuring that our approach can be easily implemented by tools. In chapter 6, I document the work we have done on developing prototype tools based on our approach, and I describe the implications of this preliminary tool development work with respect to the implementability of our approach.

Finally, it should be emphasized that these three research claims are by no means the only criteria by which our approach might be evaluated. On the contrary, there are a host of qualities that might be evaluated: the usability of the approach (which could be evaluated through user studies), whether using the approach leads to good evolution outcomes (which could be evaluated through a controlled experiment), its scalability to very large projects (which could be evaluated through simulations), and others. But the three criteria described above—applicability, computational complexity, and implementability—provide a reasonable basis for evaluation. I will discuss directions for future work in section 8.3.

1.4 Dissertation outline

The remainder of this dissertation is organized as follows. Chapter 2 details the approach to architecture evolution modeling that I illustrated informally by example in section 1.2 above. Chapter 3 undertakes a theoretical study of the specification language used to define evolution path constraints, proving some results on the computational complexity of path constraint verification. Chapters 4 and 5 describe two case studies on architecture evolution. Chapter 6 describes our preliminary work on tool implementation and reflects on what conclusions can be drawn as to the implementability of our approach to architecture evolution modeling. Chapter 7 reviews related work. Each of chapters 3–7 ends with a brief summary section explaining the chapter's significance and its relevance to the thesis statement and research claims stated in section 1.3. Chapter 8 concludes by summarizing the contributions of this thesis, identifying the limitations of our approach, and discussing directions for future work.

2 Approach

In chapter 1, I described some of the basic concepts underlying our approach and provided an informal example illustrating how these concepts are useful in planning evolution in a particular domain. In this chapter, I describe the technical basis of the approach.[2]

As mentioned previously, the approach is based on representing potential intermediate states of evolution; therefore, I begin by explaining how these *evolution states* are modeled (section 2.1). Then, section 2.2 explains the *evolution operators* that model the architectural transformations among these evolution states. Section 2.3 describes how these states and operators are used to form the nodes and edges, respectively, of an *evolution graph*. The goal of the architect, in this model, is to select the optimal path through this graph. These evolution paths can be constrained by means of *evolution path constraints*, discussed in section 2.4, and evaluated by means of *evolution analyses*, discussed in section 2.5. Finally, section 2.6 introduces the concept of an *evolution style*, which unifies these concepts and encapsulates architectural expertise about a domain of evolution in a reusable package.

Note how the elements of this approach address the research questions of section 1.1:

RQ1. *How can we model the various evolution paths available to a software architect who is planning a large-scale evolution of a system?* Sections 2.1 through 2.3 explain how we can model intermediate states of evolution, describe the architectural transformations among these states, and compose these elements to form an evolution graph with a number of candidate evolution paths.

RQ2. *How can we model and enforce constraints about which evolution paths are appropriate or permissible?* Section 2.4 describes how constraints on evolution paths can be modeled and enforced.

RQ3. *How can we facilitate analysis of evolution path quality in a way that supports architects in making trade-offs among candidate evolution paths and selecting the optimal path?* Section 2.5 describes the sorts of analysis that this approach supports and how these analyses can be used to support quality trade-offs and path selection.

RQ4. *How can we capture expertise about domains of software architecture evolution so that it can be later reused to inform future evolution planning?* Domain expertise about classes of architecture evolution can be captured by means of evolution styles, described in section 2.6.

[2]The approach detailed in this chapter was first described in a conference paper that we published in 2009 [94] and subsequently elaborated in a journal paper now in press [19].

2.1 Evolution states

One of the most basic concepts in our approach is that of the *evolution state*, an intermediate architecture that the system might take on as it evolves by stages from the initial architecture to the target architecture. There are two specially identified evolution states: the *initial state*, which captures the architecture of the system as it exists at the outset of the evolution, and the *target state*, which captures the architecture that the system is to attain by the end of the evolution. Between these two states are the transitional forms that the system may take on as it is evolved.

Each evolution state contains a complete representation of the architecture of the system. There are, of course, a variety of approaches for representing software architectures. We typically envision the architectural structure of the intermediate states as being represented using a formal architecture description language such as Acme [95]. Architecture description languages provide a rigorous way of representing the structure of a software system. Different ADLs capture different concepts, but usually an architecture is represented as a graph whose vertices represent the components of a software system (i.e., its computational elements and data stores) and whose edges represents connectors (i.e., interaction pathways among components) [53; 154; 182; 202]. In addition, there may be auxiliary elements like *ports* and *roles* (which represent interfaces of components and points of attachment between components and connectors), as well as *properties* that describe the characteristics of the architectural elements (often used to express things such as reliability and protocols of interaction), and they may support expression of constraints on and analyses over architectures.

Our approach does not impose stringent requirements on the form that an architecture description must take; in principle, the approach is compatible with almost any method for architecture description. Our approach does impose one hard requirement: there must be a way of expressing and evaluating *architectural constraints*—predicates that may hold or fail to hold for a given software architecture. This requirement is important because evolution path constraints (see section 2.4) are defined in terms of architectural constraints. Thus, without a means of expressing architectural constraints, we cannot express evolution path constraints.

Of course, the choice of architecture description language has implications for the kinds of analysis that are possible. An architecture description language that lacks a type system, for example, will obviously not permit reasoning about the types of architectural elements. Thus, adopting such a language would preclude the architect from declaring constraints such as "Until a component of type A is present in the system, no component of type B can be connected to a component of type C."

In addition, there are various practical considerations. Ideally, an architecture description language should be well supported by good tools for defining and analyzing architecture descriptions, and these tools should provide APIs or software frameworks by which the architectural models can be programmatically accessed. Chapter 6 will examine some of these implementation issues.

The bottom line, however, is that most conventional methods for architecture description are suitable for use with our approach. In fact, our approach can be used not only with formal architecture description languages like Acme, but also with other modeling languages that meet these requirements but are not generally considered to be formal ADLs, such as UML [174] and SysML [173].

In this discussion so far, I have been tacitly assuming that there is a single, canonical representation of the architecture. But often it is useful to consider multiple *architectural views.* An architectural view is a particular representation of, or perspective on, a software architecture. Clements et al. [53] identify three basic types of views: component-and-connector views (which document the structure of the running system), module views (which document the structure of the source code), and allocation views (which document the deployment and execution context of the software). Documenting a software architecture completely often requires the use of multiple views. Different projects demand different views; the choice of views used should be guided by the kinds of analysis that are needed.

Our approach allows architects to plan the evolution of a software architecture from multiple perspectives by representing multiple views of intermediate architectures [19]. This makes possible constraints and evaluation functions that make reference to multiple views. For example, "Component A shall not communicate with component B until database replicas are deployed to three separate geographic locations" is a constraint that requires examination of both a component-and-connector view and an allocation view. We could support such a constraint by including, say, a UML component diagram (representing a component-and-connector view of the architecture) and a UML deployment diagram (representing a deployment view) for each evolution state. Architects should carefully select the views to include in the evolution plan based on the analyses they anticipate; representing additional views can add significant cost to the planning process, since each view must be documented for each candidate intermediate architecture.

Finally, in addition to containing a complete architectural representation of the system (which may comprise multiple views), an evolution state may be annotated with a set of *evolution state properties.* These free-form properties may be used to supplement the evolution state with additional information as desired. For example, they could be used to demarcate specially distinguished evolution states such as release points (with a Boolean *isRelease* property), to indicate special attributes like deadlines by which transitions must occur (with a date-type property), to specify business values like the expected impact of a node on the market (perhaps with a property of some complex type), and so on.

2.2 Evolution operators and transitions

Evolution operators capture the architecture-transforming operations that are relevant to a domain of evolution. In the case of an evolution to a service-oriented architecture, there might be operators like "Wrap a legacy component as a service" and "Introduce an enterprise service bus." In the case of an evolution to cloud computing, operators might instead include "Migrate a component to the cloud" and "Migrate data from a legacy database to a cloud data store."

An evolution operator comprises: (1) a description of the *structural changes* that the operator effects; (2) *preconditions* describing the conditions under which the operator may be applied; and (3) *additional information used to support analyses*—for example, information on the cost of carrying out the evolution step or the amount of time required. Thus, the definition of an evolution operator takes the form

```
operator operatorName(parameters) {
    transformations {
        // A description of the structural changes that
        // the operator effects
    }
    preconditions {
        // The conditions under which the operator
        // may be applied
    }
    analysis {
        // Additional information to support analysis
    }
}
```

I now describe each of these parts in turn.

The structural changes of the operator are defined by means of *transformations*. A transformation represents a basic structural change to an architectural model. Thus, transformations are things like adding a component, deleting a port, renaming a connector, and modifying a property. For example, an evolution operator might be something like "Wrap a legacy component as a service," which is a single operator from an architectural standpoint but actually requires a number of transformations: introduce a new wrapper service, put the legacy component inside it, reconnect the ports appropriately, and so on. These transformations are composed to describe the structural changes that are effected by the operator.

Since transformations are basic changes to architectural *models*, the transformations available depend on the modeling language in use. In the case of Acme, the transformations will be things like adding a component, deleting a port, renaming a connector, and modifying a property. In the case of a UML deployment diagram, they will include deploying an artifact to a node and adding a communication path.

We have developed an informal, pseudocode-like operator specification language that allows for straightforward expression and composition of transformations, which is best illustrated by example. Figure 1 contains a complete definition of a *wrap-LegacyComponent* evolution operator based on the evolution scenario described in section 1.2, along with a visual depiction of its effect on an architectural diagram. The *transformations* block of the operator composes a number of transformations necessary to define the effects of wrapping a legacy component in a wrapper service. First, it creates a new wrapper service. Then, it detaches all the connectors attached to ports on the legacy component and reattaches them to ports on the wrapper. (Note the use of a *for* loop here. In addition to sequential composition of transformations, we allow simple control flow mechanisms such as loops in our operator description

```
operator wrapLegacyComponent(c) {
    transformations {
        Component wrapper = create Component
                : WrapperService;
        for (Port p : c.ports) {
            Port pw = copy p to wrapper;
            for (Role r : p.attachments) {
                detach p from r;
                attach pw to r;
            }
        }
        Representation rep
                = create Representation of wrapper;
        move c to rep;
        for (Port p : c.ports) {
            bind p to wrapper.ports[p.name];
        }
    }
    preconditions {
        declaresType(c, LegacyComponent)
    }
    analysis {
        "effortInStaffHours": 6
    }
}
```

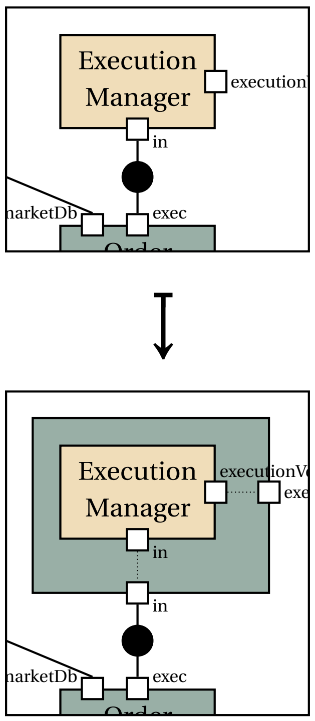


Figure 1. *An operator based on the example of section 1.2. On the left is a definition of the operator. (The architecture description language used for this example is Acme.) The diagram at right shows the effect of the operator when applied to an architecture representation.*

language.) Finally, the legacy component is moved inside the wrapper service, and its ports are bound to the corresponding ports on the wrapper service.

As I pointed out in section 2.1, sometimes we want to describe evolution from multiple architectural views. In this case, an operator must include the structural changes for each view that is under consideration. The example in figure 1 includes only one view; for examples of operator specifications containing multiple views, see our journal paper [19].

In addition to transformations, an operator can include preconditions and analysis information. *Preconditions*, which describe the conditions that must be true prior to executing the transformation, are expressed as architectural constraints. In the example of section 1.2, consider an operator that migrates a component, *c*, to the cloud. A simple precondition for this operator might be: *c* must not already be in the cloud to begin with. Of course, this is a very simple example; one could devise a more complex precondition for the same operator, for example one that imposes restrictions on the kinds of connectors to which *c* may be attached. The important point is that a precondition can be defined with respect to a single architecture at a single point; for example, it can be written as a plain Acme constraint. When multiple views are under consideration, different preconditions can be specified for each view.

The example in figure 1 contains only a single precondition, requiring that the component to be wrapped is of type *LegacyComponent*. If desired, we could add further preconditions to impose additional requirements—for example, enforcing restrictions on the types of ports that the legacy component has or the kinds of connectors to which it is attached.

Finally, analytical information may be included to support evolution analyses and constraints. For example, an operator can include information like the effort required to carry it out, the cost of carrying it out, its security implications, or the risks involved.

Analysis information is less structured than the *transformations* and *preconditions* section. The kinds of analysis information included with an operator are dependent on the analyses that the evolution style supports. To accommodate as much flexibility as possible, we allow the *analysis* block to contain arbitrary information in JavaScript Object Notation [62], which can be freely referenced by the evaluation functions described in section 2.5. The operator in figure 1 contains a single item of analytical information, an estimate of the amount of effort required to wrap a legacy component as a service. This information could be used by an evaluation function that estimates the total effort of an evolution path by adding up the effort predictions of its constituent operators. In figure 1, the effort estimate is simply specified as a number indicating a fixed number of hours of effort required, but we could also define a value that is dependent on the parameters of the operator, or we could specify a range of values to indicate uncertainty.

Given a set of operators defined in this way, we can use them to define *evolution transitions* among the evolution states. An evolution transition is composed of one or several operators and describes the architectural transformations necessary to evolve the system from one state to another. Like an evolution state, an evolution transition can be annotated with one or more properties that provide supplementary information (for example, the number of personnel available to carry out the transition).

2.3 The evolution graph

The task of the software architect is to develop a plan for evolving the system from its initial architecture to a desired target architecture. We can describe such a plan as a series of evolution states and evolution transitions leading from the initial architecture to the target architecture. An *evolution path* is a plan described in such a way.

In a given evolution scenario, there may be many evolution paths under consideration. To understand the space of alternatives under consideration, we construct

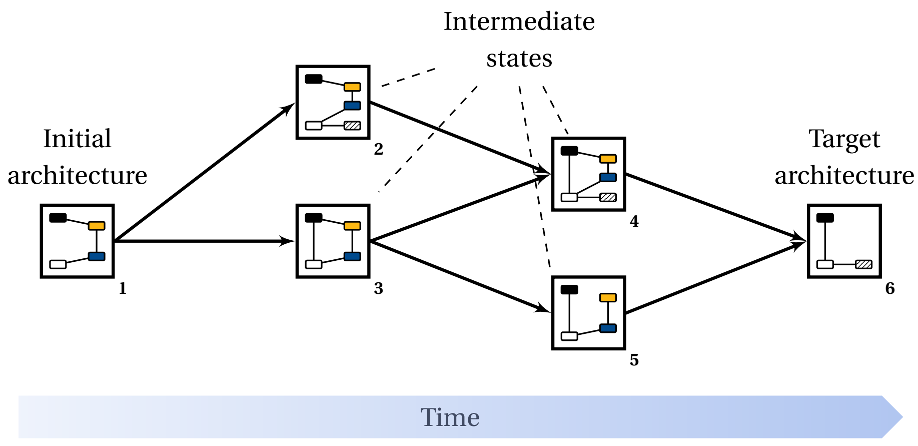


Figure 2. *A depiction of an evolution graph. Each node in the graph is a complete architectural representation of the system. The edges of the graph represent possible evolutionary transitions. The architect's task is to select the optimal path through the graph. This graph has only three paths: 1–2–4–6, 1–3–4–6, and 1–3–5–6.*

an *evolution graph*—a graph whose nodes are evolution states and whose edges are evolution transitions. This concept is illustrated in figure 2.

The task of the software architect is to select an optimal *evolution path*—a way of evolving the system from the initial architecture to the target architecture. An evolution path thus captures a particular plan of evolution. Given an evolution graph as defined above, an evolution path is simply a path in the graph-theoretical sense: an open walk from the node corresponding to the initial architecture to that corresponding to the target architecture. In any given evolution scenario, there may be many evolution paths that seem feasible; the task of the evolution architect is to consider these paths and select the one that is optimal. I will clarify the meaning of *optimal* in section 2.5.

This graph-theoretical formulation of software architecture evolution is a simple one, but it provides a useful foundation for reasoning about potential evolution plans and the distinctions among them. In particular, it gives us a basis for reasoning about:

- Intermediate states of evolution and the architectural distinctions among them
- Release points, milestones, and other specially distinguished stages of evolution
- Evolution paths in terms of the individual steps that compose them
- Transitions among candidate intermediate architectures, including their architectural effects and their properties
- Constraints expressing rules about the evolution domain (such as architectural dependencies, restrictions on releases, invariants, etc.)
- Path utility (and other properties of paths such as expected evolution duration and cost)

- Trade-offs among evolution paths

We have already seen how some of these concepts are modeled. The next two sections discuss constraints and evaluation functions.

2.4 Evolution path constraints

Path constraints capture evolution path correctness; an evolution path may be regarded as correct or legal if it satisfies all defined constraints. Examples of evolution path constraints from various domains are:

1. Once a server is placed in the Boston data center, it must not be removed (domain: data migration).
2. The billing subsystem will not be removed until a controller component is introduced (domain: e-commerce).
3. All the services that are present at a release point remain present throughout the evolution (domain: service-oriented architecture).
4. As a web host's server farm is upgraded, hosting operations must continue uninterrupted, with no disruption to clients (domain: in-place evolution in a server farm).

Once appropriate constraints for a domain are defined, many of the paths in an evolution graph can be eliminated automatically by constraint enforcement. This simplifies path selection.

Formally, a path constraint is a predicate (Boolean-valued function) over evolution paths. Thus, a constraint judges each evolution path to be either valid or invalid with respect to the rule of evolution that the constraint encodes.

Path constraints may be specified in an augmented version of linear temporal logic (LTL). Temporal logic is a natural choice, since path constraints are temporal propositions. In addition, the interpretation of temporal formulas with respect to an evolution graph is straightforward. An evolution graph can be naturally viewed as a Kripke structure whose states are simply the evolution states and whose transitions are simply the evolution transitions. This allows temporal formulas to be defined and interpreted in the usual way.

Chapter 3 will describe our path constraint specification language in full detail, but here I give a brief introduction. We begin with the usual LTL operators, including:

- $\square$, *always*, to represent an invariant property of a path;
- $\lozenge$, *eventually*, to represent the existence in a path of an architecture with certain properties;
- $\mathbf{U}$, *until*, to represent properties that must remain true of a path until some other property becomes true; and
- $\bigcirc$, *next*, to represent properties that must be true in the next step of the path.

Although LTL is a good start—there are many path constraints that can be expressed in LTL in a straightforward manner—it turns out that some path constraints are inexpressible in LTL. For example, consider constraint 3 from the list above. If we try to express this constraint in LTL, we quickly encounter a problem. We might start by trying something like

$$\Box(\mathit{release} \rightarrow \Box \mathit{hasAllServices}(\mathit{system}, \underline{\qquad})). \tag{1}$$

(In this example, *release* is a property of evolution nodes, *hasAllServices* is a user-defined predicate expressing that one architecture has all the same services as another, and *system* is a keyword that refers to the architecture description associated with a state.)

The problem is that to express this constraint, we need to refer back to a previous state, namely the previous release. That is, we want to replace the blank in equation (1) with a reference to the previous release state. LTL, however, cannot capture predicates relating multiple different states. This can be solved by extending LTL with a rigid-variable operator, which allows us to refer directly to states that we have already "seen." In our notation, equation (1) would be correctly rendered as

$$\Box(\{s\}\ \mathit{release} \rightarrow \Box \mathit{hasAllServices}(\mathit{system}, s.\mathit{system})).$$

The braces are our rigid-variable operator. When we encounter them, they "save" the current state to the rigid variable *s* so that we can refer back to it as such in a subsequent step. Because of the finite nature of paths, it is possible to check whether a given evolution path satisfies a given set of evolution constraints. Thus verification of path constraints can be automated. Chapter 3 will treat the path constraint language in detail, discussing a number of related logics and proving results about the computational complexity of path constraint verification.

Figure 3 shows a constraint based on the example of section 1.2, illustrating how a constraint is verified with respect to an evolution path.

2.5 Evolution path evaluation functions

With the aid of constraints, an architect can define paths that are technically correct, taking into consideration rules about what constitutes a valid path. However, the real benefit of defining paths of evolution is to compare them and decide which path is the best to take. This is the purpose of *evolution path evaluation functions*.

Where path constraints provide a hard, yes-or-no judgment about the correctness of a path, path evaluation functions instead provide a quantitative judgment about its goodness. Formally, constraints and evaluation functions are close kin: an evolution constraint is a Boolean-valued function (i.e., a predicate) over paths, while an evaluation function is a numeric-valued function. They tend to be used differently, however. Constraints are useful for expressing basic rules about a domain; their ultimate purpose is to prune the evolution graph so that it is more manageable. Evaluation functions are useful for judging the quality of candidate evolution plans; their ultimate purpose is to aid the architect in selecting a specific path by facilitating estimation of path quality. Figure 4 illustrates this concept.

Constraint: Once a client is connected to the new dashboard,
no further modifications can be made to elements in the cloud.

$\Box(\{s\}\ \mathit{clientConnectedToUnifiedDashboard} \rightarrow \Box \mathit{cloudUnchanged}(\mathit{system}, s.\mathit{system}))$

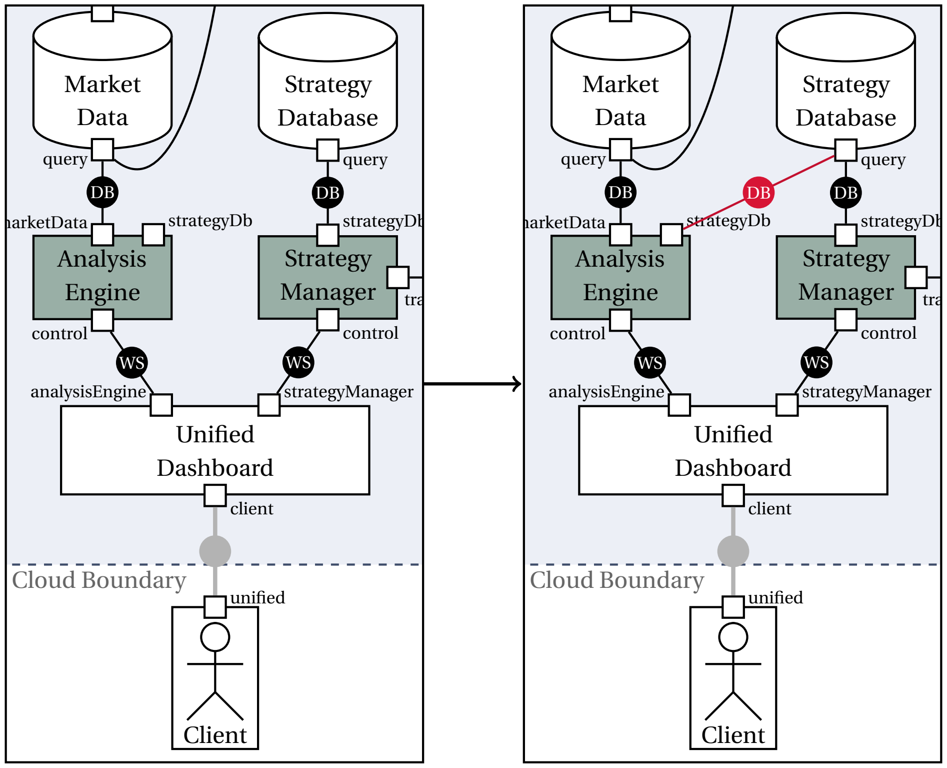


Figure 3. *Illustration of how a particular constraint is evaluated on a particular evolution path. At the top are a path constraint (based on the example of section 1.2) and its formalization. The diagram depicts a fragment of an evolution path that violates the constraint. In this evolution path, a connector (shown in red) is added between the "Analysis Engine" and "Strategy Database" components after a client is already connected to the dashboard, violating the prohibition on modifying the cloud while a client is attached. Not shown are the definitions of clientConnectedToUnifiedDashboard and cloudUnchanged, which are predicates over architecture descriptions and can be defined by conventional means (e.g., using Acme).*

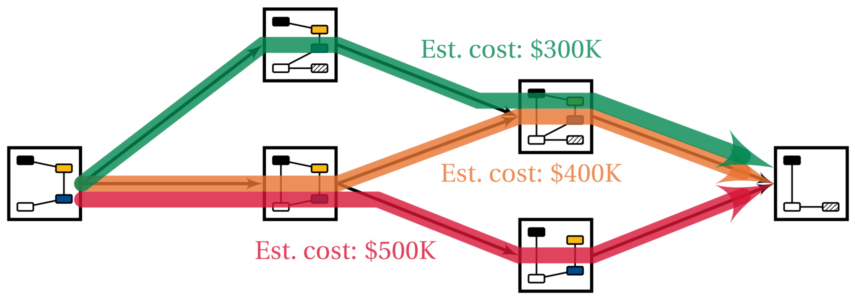


Figure 4. *Evaluation functions facilitate path selection by assigning to each evolution path a value (in this case, an estimate of the total cost of the path).*

Evaluation functions help software architects to determine whether a path satisfies business and management goals. In general, evaluation functions will depend on attributes specific to a particular business context: (a) the qualities of concern (cost, functionality, time, etc.) and their relative priorities, and (b) constraints on resources (number of personnel, time to deliver a release, etc.).

Ultimately, the goal of evolution graph analysis is to select an optimal path. This requires the definition of an evaluation function that provides an evaluation of the overall goodness of each path, so that the paths may be compared. Since different business contexts have different conceptions of "goodness," we must define a measure of path *utility* that is appropriate for the domain at hand. Thus, an evaluation function for utility might be a composite of evaluation functions for attributes such as time, cost, and risk, chosen and weighted appropriately for the problem under consideration.

Another way of developing a utility function is with a cost-benefit approach. Typically, it makes sense to associate costs with transitions—representing the effort, time, money, and other resources that will be expended in carrying out each transition—and benefits with nodes—representing the expected utility from releasing new versions of the system.

Utility functions allow trade-off analysis. For example, one evolution path might feature a steady evolution with a large number of incremental releases, while another might have comparatively few releases, each with major changes—requiring the investment of substantial resources but reducing the time to reach the target architecture [39]. By using evaluation functions to explore this utility space, the architect can make this time/cost trade-off intelligently.

We have not defined a special-purpose specification language for evaluation functions. Instead, we assume that evaluation functions are defined as functions in a general-purpose programming language, which take the evolution model as input and produce a numeric value as output. In our work, we have used JavaScript (or, more precisely, ECMAScript [77]) as our language for defining evaluation functions, due to its ubiquity and readability, but any language can be used in principle. Examples of definitions of evaluation functions will be presented in chapter 5.

2.6 Evolution styles

Many of the concepts I have mentioned are domain-specific. For example, in the domain of evolution to a service-oriented architecture, there might be SOA-relevant operators like “Wrap a legacy component as a service,” SOA-relevant constraints like “Once an enterprise service bus is introduced, all service components must communicate via it," and SOA-relevant evaluation functions based on domain expertise about the costs and benefits of various SOA operators. Indeed, many of the concerns that arise during evolution planning are domain-specific. Domain expertise can aid significantly in planning evolution, and so domain specialization is an important part of the approach proposed here.

Domain specialization is achieved by means of *evolution styles*, which encapsulate architectural expertise about a domain of software evolution. An evolution style provides common properties, operators, constraints, and evaluation functions to support modeling and reasoning about a broad class of evolutions. For example, consider the following domains of evolution:

1. Evolutions from a thin-client/mainframe system to a tiered web application
2. Evolutions from a J2EE web services architecture to a cloud-computing architecture
3. Evolutions to improve the security of a service-oriented architecture
4. Evolutions from a legacy version of IBM WebSphere to the most recent version

Each of these examples refers to a class of evolutions addressing a recurring, domain-specific architecture evolution problem. Each of them has identifiable starting and ending conditions (namely, that the initial and final system adhere to certain architectural styles, contain certain architectural structures, or have certain architectural properties). Each embodies certain constraints—for example, that the set of essential services should not become unavailable during the evolution. Finally, although they share many commonalities, the specific details of how those evolutions should be carried out may well be influenced by concerns such as the time it takes to do the transformation, the available resources to carry it out, etc. We can take advantage of these characteristics of system evolution.

Evolution styles are defined by analogy with the traditional concept of *architectural styles* in software architecture. An architectural style is a class of architectures that share common element types and properties, such as the pipe-and-filter architectural style and the service-oriented architecture style. An architectural style is defined by a vocabulary of architectural element types, together with a set of constraints that govern how instances of those types can be composed into systems [202]. In a similar way, an evolution style defines a vocabulary of evolution operators, together with constraints governing how they may be composed into paths. More precisely, an evolution style comprises:

- A characterization of the domain to which the evolution style applies (e.g., evolutions from a J2EE web services architecture to a cloud-computing architecture, or evolutions whose goal is to enhance the security of a service-oriented architecture)

- A set of evolution operators describing architectural transformations relevant to the domain
- A set of path constraints to define the rules of the domain
- A set of evaluation functions relevant to the domain

3 Theoretical results on evolution path constraint verification

Chapters 4 and 5 will undertake an empirical evaluation of the approach just described, evaluating its applicability to real-world evolution scenarios. Before turning to this empirical work, however, it is useful to conduct a *theoretical* analysis of some of the properties of our modeling apparatus. In this chapter, therefore, I evaluate our path constraint language by establishing a theoretical foundation for it and then using this foundation to evaluate its tractability (more precisely, the computational complexity of evaluating a path constraint).[3]

I focus on the path constraint language in particular because it is the most theoretically interesting part of our modeling apparatus. Other parts are either trivial and uninteresting (e.g., operator preconditions and local judgments about architectural styles) or too general to say anything about (e.g., evaluation functions, which provide the evolution planner with the full power of a general-purpose programming language, so the complexity of evaluation is entirely dependent on the complexity of the evaluation function that the planner chooses to write).

Sections 3.1 and 3.2 present a formal syntax and semantics, respectively, for the path constraint language. Section 3.3, places this language in context by discussing a number of other, similar logics that have likewise been formed by supplementing LTL with a variable-binding operator and identifies where the important differences lie. Section 3.4 contains the complexity result. Finally, section 3.5 summarizes the chapter and discusses the significance of the main results.

3.1 Syntax of the constraint specification language

Section 2.4 introduced the path constraint specification language informally. As I said there, it is based on LTL, which has a very simple syntax:

$$\phi := p \mid \mathit{false} \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc\phi \mid \phi_1 \,\mathbf{U}\, \phi_2$$

for propositional symbols p. Other connectives that appear in LTL can be defined in terms of these, for example $\neg\phi := \phi \rightarrow \mathit{false}$ and $\Diamond\phi := \mathit{true}\,\mathbf{U}\,\phi$ and $\Box\phi := \neg\Diamond\neg\phi$.

Such a simple definition will not work for our evolution path constraint language. We need to be concerned not only with ordinary propositions, but also with predicates and functions. A condition such as "The software architecture has at least one

[3] Much of the material in this chapter is adapted from the theoretical discussion in our recent journal article [19, § 4].

component" could be represented as a proposition p. But a more interesting condition such as "This software architecture has at least the same database components as the one in the previous state" expresses a relation over two different architectures. So in defining a syntax, we need to recognize that there are atomic formulas other than merely propositional symbols. In this respect, it is similar to first-order predicate logic (FOL), and so in formalizing the syntax for our path constraint language, we take a cue from FOL, giving separate, inductive definitions for terms, atomic formulas, and finally formulas:

Definition 1 (path constraint syntax)**.** Let V be a set of variables. For $n = 0, 1, 2, \ldots$, let F_n be a set of n-ary function symbols and let P_n be a set of n-ary predicate (relation) symbols. Together these sets form a *signature*, which we write as $\Sigma = (V, (F_n)_{n\in\mathbb{N}}, (P_n)_{n\in\mathbb{N}})$. (This is just as in FOL.)

The terms π, atomic formulas α, and formulas ϕ are defined inductively:

$$\begin{aligned}
&\pi := x \mid f(\pi_1, \ldots, \pi_n)\\
&\alpha := p(\pi_1, \ldots, \pi_n)\\
&\phi := \alpha \mid \mathit{false} \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc\phi \mid \phi_1 \,\mathbf{U}\, \phi_2 \mid \{x\}\phi
\end{aligned}$$

for $x \in V$, $f \in F_n$, and $p \in P_n$.

To make this definition more concrete, let us analyze the example formula from section 2.4 in terms of this syntax. In the formula

$$\Box(\{s\}\ \mathit{release} \rightarrow \Box\mathit{hasAllServices}(\mathit{system}, s.\mathit{system})),$$

hasAllServices is a binary predicate and *release* is a proposition (nullary predicate).

The keyword *system* is a function. When it appears alone, as in the first occurrence of *system* in the formula above, is is a nullary function: it takes no arguments and behaves as a term. (This might be surprising, since in FOL we often refer to nullary functions as "constants," and *system* does not hold constant—on the contrary, it refers to something different in every state. But in a temporal context, *constant* is not a very good word for a nullary function, because a nullary function can refer to different states depending on the current state, just as a nullary predicate—a proposition—can have a different truth value from state to state. For a different approach, see half-order modal logic [108], where functions have a "rigid"—state-independent—interpretation and predicates have a "flexible" interpretation.) When *system* is applied to the variable s, it is a unary function.

Where do the predicates come from? Where are things like *hasAllServices* defined? The answer was prefigured in section 2.1, which noted that in order to be suitable for use with our approach, an architecture modeling language must provide a means of specifying architectural constraints—predicates over architecture descriptions. The predicates in our temporal path constraints, then, can be specified as conventional architectural constraints, using the facilities provided by the architecture modeling language. The Acme ADL, for example, has a constraint language called Armani [159], with which we can define architectural predicates such as *hasAllServices*. We will see examples in section 5.4.3.

The function *system*, on the other hand, is a keyword built into the path constraint language. This is the only predefined function that currently exists in the path constraint language—in our experiences defining path constraints thus far, it has been sufficient on its own for expressing constraints that are of interest—but other predefined functions (or user-defined functions) could be added in principle without affecting the formalization in this chapter, such as a unary *successor* function that returns the next intermediate architecture in the temporal sequence.

3.2 Semantics of the constraint specification language

I will now give a semantics for our path constraint language. Let us begin by recalling the semantics of LTL. There are various ways to formalize the semantics of LTL. In the following formalization, we identify a state with an interpretation of the propositional symbols. Alternatively we could externalize an interpretation function as a map from states to sets of propositions.

The Kripke semantics for LTL is as follows. Let P be a set of propositional symbols. Let σ be a sequence of states: $\sigma_1, \sigma_2, \ldots$, where $\sigma_i \subseteq P$ for each i. (Thus, each state comprises the set of propositional symbols that are interpreted to hold true in that state.) We write $\sigma, i \vDash \phi$ to say that σ satisfies the LTL formula ϕ at a time $i > 0$. We define this satisfaction relation inductively:

- $\sigma, i \vDash p$ iff $p \in \sigma_i$ (i.e., iff the propositional letter p is true under the interpretation given by σ_i).
- $\sigma, i \vDash \mathit{false}$ never holds.
- $\sigma, i \vDash \phi \to \psi$ iff $\sigma, i \vDash \phi$ implies $\sigma, i \vDash \psi$.
- $\sigma, i \vDash \bigcirc\phi$ iff $\sigma, i+1 \vDash \phi$.
- $\sigma, i \vDash \phi \mathbf{U} \psi$ iff there is some $j \geq i$ with $\sigma, j \vDash \psi$ such that $\sigma, k \vDash \phi$ whenever $i \leq k < j$.

Recall from section 3.1 that the other connectives, such as $\neg$ and $\Diamond$ and $\Box$, can be defined in terms of these, so there is no need to give a semantics for them.

There are a number of things we must change to obtain a semantics for our path constraint language. First, LTL normally models a sequence of infinitely many states, $\sigma_1, \sigma_2, \ldots$. We, however, are interested in expressing constraints over a finite sequence of states: the evolution path, which comprises finitely many intermediate software architectures. So the first modification we will need to make to the definition of the semantics of LTL is to restrict ourselves to a finite sequence of states. (After all, an evolution plan that requires infinitely many operations to be carried out to reach the target architecture is not a very useful evolution plan!)

The second change we need to make is to account for our additions to the syntax. Atomic terms are much richer than they are in LTL. Again we take a cue from FOL. In propositional logic, an interpretation is simply an assignment of truth or falsehood to each proposition symbol. But in FOL, an interpretation is a map that assigns a function to each function symbol and a relation to each predicate symbol. Similarly, in our path constraint semantics, a state now needs to be more than simply an identification of which propositional letters are true; it should be an interpretation function that maps the function and predicate symbols of the syntax to functions and relations. (In FOL, these are functions and relations on the domain of quantification; for us they are functions and relations on the temporal states.)

Finally, we need to express the semantics of our new variable-binding operator, which means we must keep track of what states the variables are binding to. This is not as straightforward as it sounds; just adding a line to the definition of the satisfaction relation does not work. We also need a way to keep track of what states the variables are binding to. After making all these changes, we obtain the following semantics.

Definition 2 (path constraint semantics)**.** Let

$$\Sigma = (V, (F_n)_{n\in\mathbb{N}}, (P_n)_{n\in\mathbb{N}})$$

be a signature. Let σ be a sequence of states, $\sigma_1, \sigma_2, \ldots, \sigma_n$. As in LTL, we define a *state* to be an interpretation, but now each state σ_i is a function that maps each function symbol to a function over states and each predicate symbol to a relation over states. That is,

- If $f : F_n$, then $\sigma_i(f) : S^n \to S$, where S is the set of states.
- If $p : P_n$, then $\sigma_i(p) \subseteq S^n$.

A *variable assignment* $s : V \to S$ is a function that maps variables to states. (This is needed to keep track of what free variables stand for.) We write $\sigma, i, s \vDash \phi$ to say that σ satisfies the path constraint ϕ at time $i \in \{1, \ldots, n\}$ under the assignment s. We define the *denotation* of a term π in the structure σ at time i under assignment s, written $D_{\sigma,i,s}(\pi)$, by

- $D_{\sigma,i,s}(x) := s(x)$
- $D_{\sigma,i,s}(f(\pi_1, \ldots, \pi_n)) := (\sigma_i(f))(D_{\sigma,i,s}(\pi_1), \ldots, D_{\sigma,i,s}(\pi_n))$

Finally, we define the satisfaction relation inductively:

- $\sigma, i, s \vDash p(\pi_1, \ldots, \pi_n)$ iff $(D_{\sigma,i,s}(\pi_1), \ldots, D_{\sigma,i,s}(\pi_n)) \in \sigma_i(p)$
- $\sigma, i, s \vDash$ *false* never holds.
- $\sigma, i, s \vDash \phi \to \psi$ iff $\sigma, i, s \vDash \phi$ implies $\sigma, i, s \vDash \psi$.
- $\sigma, i, s \vDash \bigcirc\phi$ iff $i = n$ or $\sigma, i+1, s \vDash \phi$.[4]
- $\sigma, i, s \vDash \phi \,\mathbf{U}\, \psi$ iff there is some $j \in \{i, i+1, \ldots, n\}$ with $\sigma, j, s \vDash \psi$ such that $\sigma, k, s \vDash \phi$ whenever $i \le k < j$.
- $\sigma, i, s \vDash \{x\}\phi$ iff $\sigma, i, s[x \mapsto \sigma_i] \vDash \phi$ (where $s[x \mapsto \sigma_i]$ is the same assignment as s except with x now assigned to σ_i)

If ϕ is a closed sentence (i.e., has no free variables) then the assignment s is irrelevant and we may simply write $\sigma, i \vDash \phi$.

[4]This is "weak next," meaning that $\bigcirc\phi$ is interpreted to be true in the final state of a sequence. "Strong next" can be defined in terms of the weak next operator: $\overline{\bigcirc}\phi := \neg\bigcirc\neg\phi$.

In the end, we get something that looks more like the semantics of FOL than that of LTL. Indeed, as we will see in the next section, some authors who have introduced similar logics have referred to such a variable-binding construct as a special kind of quantifier; this formalization shows why that is appropriate.

3.3 Similar logics

It is important to understand how our path constraint language relates to the existing landscape of temporal logics, both to provide a context of related work and to clarify the significance of the results in section 3.4. Indeed, there are several logics that are similar to ours with respect to our introduction into LTL of variables that retain their values across states.

A logic that is very similar to ours is one developed by Richardson [187] to support lists in an object-oriented data model. Richardson's logic is essentially LTL with the addition of "rigid variables," which are semantically identical to our extension to LTL. Richardson's logic does not appear to have been studied for its theoretical properties; the paper has not been widely cited outside the database community. "Rigid variables" appear elsewhere in the literature as well, most famously Lamport's temporal logic of actions [126], although Lamport's rigid variables are somewhat different.

Another related logic has been developed to model real-time systems. A natural way to specify real-time systems is with LTL, but one problem that arises is the incorporation of hard real-time requirements. Alur & Henzinger [4] developed a logic that they called timed propositional temporal logic (TPTL), whose main feature was the introduction of *freeze quantifiers*, which bind a variable to a particular time so that it can be accessed later. These are similar to our rigid variables. There are a couple of differences, which I describe under the next subhead.

Also very similar is Lisitsa & Potapov's temporal logic with predicate λ-abstraction [140]. Lisitsa & Potapov's work is a specialization of the abstraction device for modal logic documented by Fitting [83] to LTL. This abstraction mechanism is the same variable-capturing device as our rigid variables or Alur & Henzinger's freeze quantifiers, although it appears to have been developed independently. As Demri et al. [71] observe, "Even though this construct is essentially the same as the freeze quantifier, apparently there have been no cross-references between the literature dealing with predicate λ-abstraction [...] and that dealing with the freeze quantifier [...]."

Yet another related logic is Goranko's temporal logic with reference pointers [102], which differs in a couple of ways. First, unlike Richardson, Alur and Henzinger, and us, who were devising specification languages for particular domains (object-oriented data models, real-time systems, and software architecture, respectively), Goranko is philosophically motivated. He notes that LTL lacks a way to refer to particular points in time—to express the concept "then." Unlike the other logics we have seen, which give explicit names to states, Goranko's logic simply uses the symbol $\downarrow$ to indicate a point that we might refer to later, then uses $\uparrow$ to refer to it (to say "then"). Syntactically, $\uparrow$ behaves like a propositional variable; semantically, $\uparrow$ is true if the current time is the same as the time of $\downarrow$. Goranko uses this to express things like "now will not occur again": $\downarrow\Box\neg\uparrow$.

A final related family of logics is hybrid logic [29; 30], where states can be referred to via labels called *nominals*. A nominal is an atomic symbol with the special property that it is true at exactly one state. We can also use a nominal a to build *satisfaction statements*, which have the form $@_a\phi$, which means "ϕ is true relative to the state characterized by a." Finally, often hybrid logics are supplemented with a $\downarrow$ binder, which binds a label to the current state, much like our rigid variables (or a named version of Goranko's $\downarrow$). The result is powerful. For example, we can now define *until* in terms of these hybrid logic constructs:

$$\phi \text{ until } \psi \quad := \quad \downarrow x(\Diamond \downarrow y(\psi \wedge @_x\Box(\Diamond y \rightarrow \phi)))$$

Hybrid logics are rather different from our logic, but the basic idea of named states, and in particular the $\downarrow$ binder, are closely related.

This is not an exhaustive list; Blackburn and Tzakova [30], for example, cite a few others, observing, "Labeling the here-and-now seems to be an important operation." Indeed, the idea seems to have been reinvented numerous times. Another discussion of related logics is given by Demri et al. [71, § 5].

How our evolution path constraint language differs. Our language fits comfortably within this family of related logics. Operators that bind variables are nothing new. However, there are some ways in which our path constraint language distinguishes itself semantically from its cousins. Although these distinctions are subtle, they turn out to have, in some cases, major theoretical consequences. Although the existing literature is rich, it is also somewhat patchy. There are interesting problems that have yet to be tackled. Notably, the question of the complexity of *model-checking a path*, which we discuss below and resolve in section 3.4, is a natural one that has been solved for LTL and a number of extensions to LTL [147] but not for LTL with a variable-binding operator. In the remainder of this section, I will discuss the differences between our path constraint language and other logics. I will focus on the two related logics that are the most mature and best studied: TPTL and hybrid logic.

TPTL was invented by Alur and Henzinger to model real-time systems, but (likely due to their extensive theoretical characterization of their new "freeze quantifier" and their generalization of the idea beyond their domain) their work became quite influential outside of this field. The semantics of TPTL differs from our path constraint language in two important ways. First, TPTL, like LTL, assumes an infinite sequence of states; our logic assumes a finite sequence. Second, the variables that freeze quantifiers capture are times (natural numbers) rather than architectural models. All they do with the variables they freeze is compare them to other times with operators like $\leq$; we want to do architectural analysis. At first glance, these seem to be peripheral issues, but they turn out to be important. Indeed, Alur and Henzinger themselves showed that small changes to the language can substantially change its theoretical properties; for example, supplementing TPTL with addition over time renders the satisfiability problem highly undecidable.

The results that I will present in section 3.4 agree with this generalization. These seemingly subtle semantic changes have substantial ramifications on the theoretical properties of the language, and not always in the way that one would expect. For example, consider the problem of *model-checking a path*: evaluating whether a formula holds for a single path. Alur and Henzinger show that this problem is EXPSPACE-complete for TPTL, further noting that it would be undecidable if TPTL were modified to allow more powerful atomic propositions, such as addition over terms. Our path constraint language, on the other hand, goes so far as to allow atomic propositions made up of *arbitrary* predicates over arbitrary terms (just like FOL), but then we regain decidability by studying finite sequences rather than infinite sequences. Unsurprisingly, this completely changes the problem of model-checking a path. What is perhaps surprising is that the problem is still hard and interesting. The problem of model-checking a single, finite path might be naively thought to be rather trivial, but Markey and Schnoebelen [147] suggest that in fact the problem is of substantial theoretical interest. The results in section 3.4 support their contention. As I will show, the problem of model-checking an evolution path constraint on a finite path is hard (PSPACE-complete). Not only that, but this complexity result does not lend itself to easy proof; a fairly sophisticated reduction strategy was necessary to prove PSPACE-hardness.

Much the same can be said of hybrid logic. Although hybrid logic *can* be used to express constraints over finite, linear paths, it does not seem to be done often, at least not often enough that anyone has bothered to study the theoretical properties of that case. And again, although naively we might assume this case to be trivial, boring, or a straightforward specialization of the more general case, in fact interesting and surprising results emerge from these restrictions.

Hybrid logic is quite different in character and focus from our constraint logic. In particular, although the general idea of *nominals* (propositions that are true in only one state and hence uniquely identify that state) is quite central to hybrid logic, the $\downarrow$ binder that corresponds to our variable-binding operator is not. Rather, $\downarrow$ was a late-breaking addition, imported into hybrid logic from Goranko's temporal logic of reference pointers [102]. This is not to say that the theory of $\downarrow$ in hybrid logic is underdeveloped—on the contrary, some remarkable results about it have been published—but the focus of hybrid logic is different from ours. More characteristic of hybrid logic than $\downarrow$ is the aforementioned *satisfaction operator*; it effectively jumps to a named state. Not having the satisfaction operator imposes some unexpected challenges. For example, as I note in section 3.4, having the satisfaction operator would make proving the PSPACE-hardness result dramatically less challenging.

Of course, nominals themselves are also somewhat different from bound variables in our language. A nominal is a proposition that is true in exactly one state; a bound variable is a *term* that directly captures a state object. This is an important semantic difference, although the effect is similar.

3.4 Computational complexity

The primary thing we want to do with path constraints, of course, is check whether a given path satisfies a given constraint. This can be easily stated as a model-checking problem. In general, model checking is the problem of checking whether a specific formula is true of a specific Kripke structure. More specifically, model checking is usually used to verify that a state transition system has some property. For some logics, such as CTL, this is easy. For others, such as LTL, it is hard—PSPACE-complete, in fact. That is, given a finite state transition system, determining whether an LTL formula is valid in that transition system is PSPACE-complete [205]. Moreover, the solution to the model-checking problem for LTL is intellectually rather challenging too, involving an intricate tableau construction. There is certainly not much hope that our variable-binding construct will make things any easier, especially in light of the result that model-checking TPTL is EXPSPACE-hard [4].

Fortunately, we are not terribly interested in this form of the problem. Instead, we are interested in model-checking a single, particular path—not verifying a formula over an entire state transition system. Our primary use case is telling software architects whether the paths that they have planned are admissible according to the constraints; we do not need to check all the paths in some transition system, nor even a great number of paths—just one, or a few, at a time. Likewise, all our paths are finite—and in most cases probably rather short, since they must be comprehensible to the human architects reasoning about them.

Model-checking a single path is a much easier problem computationally, but one that has been recognized in recent years as theoretically interesting [147].[5] For pure, propositional LTL, model-checking a formula of length ℓ on a path of length m takes $O(\ell m)$. The algorithm for single-path model checking for LTL is the same as the familiar algorithm for model-checking CTL, since LTL and CTL coincide over individual paths [147]. This traditional algorithm [52] uses a dynamic-programming approach. To determine whether $\sigma, i \vDash \phi$ for some finite temporal sequence σ and formula ϕ, we list the subformulas of ϕ and solve $\sigma, i \vDash \phi_i$ for each subformula ϕ_i. We begin with the smallest subformulas (i.e., atomic formulas), which are immediately solvable, then inductively solve larger subformulas by using the solved subformulas that compose them.

Things become messier when we add the variable-binding operator, $\{x\}$. This simple dynamic-programming algorithm is no longer adequate, because we also need to keep track of variable valuations. For example, in the formula $\Box\{x\}\Box p(x)$, the truth of $p(x)$ depends not only on the state at which we are evaluating $p(x)$, but also on the value of x. To determine whether the formula holds on a path, we ultimately need to evaluate $p(x)$ at each state *and for each value of* x. The subformula $p(x)$ thus needs to be evaluated $O(m^2)$ times; in model-checking finite LTL paths, we never need to evaluate a subformula more than m times. So it is clear that model-checking our path constraint language will be harder than model-checking finite LTL paths. But how hard?

Consider a formula of the form

$$\Box\{x_1\}\Box\{x_2\}\cdots\Box\{x_k\}\Box p(x_1,\ldots,x_k)$$

[5] In the program verification community, the problem of checking a finite, single trace is known as *runtime verification*, with the term *model checking* reserved for checking entire state structures [22, § 1.1].

The only apparent way to check this formula is to evaluate $p(x_1,\ldots,x_k)$ at each state and for each valuation $x_1 \le \cdots \le x_k$. There are $\binom{m+k-1}{k}$ such valuations, which is $\Theta(m^k)$ for fixed m. Note that k, the number of rigid variables, is asymptotically proportional to the length of the formula, ℓ. Thus, we have to evaluate $p(x_1,\ldots,x_k)$ under $\Theta(m^\ell)$ different valuations. This gives us a strong hint that we have departed the realm of polynomial-time algorithms.

However, we can model-check a path constraint in polynomial *space*, because we only need to work with one valuation at a time. In fact, even the naive recursive algorithm is polynomial-space:

Theorem 1. *There is a polynomial-space algorithm for model-checking a path constraint on a single path.*

Proof. Let σ be a temporal sequence of finite length m. Let ϕ be a path constraint of length ℓ. We define a recursive algorithm to determine whether $\sigma, i \vDash \phi$:

```
CHECK($\sigma, i, \phi, s$)
    if $\phi$ is an atomic proposition
        evaluate $\phi$ at $i$ using assignment $s$
    if $\phi = false$
        return false
    if $\phi$ is of the form $\chi \to \psi$
        return CHECK($\sigma, i, \psi, s$) or not CHECK($\sigma, i, \chi, s$)
    if $\phi$ is of the form $\bigcirc\psi$
        return $i = m$ or CHECK($\sigma, i+1, \psi, s$)
    if $\phi$ is of the form $\chi \mathbf{U} \psi$
        for $j = i$ to $m$
            if CHECK($\sigma, j, \psi, s$)
                return true
            if not CHECK($\sigma, j, \chi, s$)
                return false
        return false
    if $\phi$ is of the form $\{x\}\psi$
        return CHECK($\sigma, i, \psi, s[x \mapsto \sigma_i]$)
```

To check whether $\sigma, i \vDash \phi$, we call CHECK($\sigma, i, \phi, \emptyset$).

Consider the space complexity of this algorithm. Since every recursive call is of a strict subformula, the stack depth of this algorithm never exceeds ℓ. Each execution of CHECK (excluding the recursion) uses only $O(1)$ space. Thus, the space complexity of the algorithm is $O(\ell)$. □

This shows that the model-checking problem for path constraints is in PSPACE. I will now prove that it is PSPACE-complete. The way to prove that a problem is PSPACE-hard is to prove that the quantified-Boolean-formula (QBF) problem reduces to it. QBF is the canonical PSPACE-complete problem; it fulfills the same role for PSPACE that the Boolean satisfiability problem (SAT) fulfills for NP. In fact QBF is a direct generalization of SAT. A quantified Boolean formula is just what it sounds like: a formula such as

$$\exists x \forall y \exists z (z \wedge x \vee y) \tag{2}$$

where each variable is interpreted as Boolean and is quantified. (This QBF is in prenex normal form: it comprises a string of quantifiers followed by a quantifier-free propositional form. Any QBF can be converted to prenex normal form in polynomial time, so from now on we will assume prenex normal form, as is typical in QBF proofs.) The QBF problem is to determine whether the formula is true. SAT can be viewed as the special case of QBF where all the quantifiers are existential.

There is a fairly obvious transformation from QBF to our model-checking problem, but unfortunately this obvious reduction does not work. The obvious reduction is to change the $\forall$s into $\Box$s and the $\exists$s into $\Diamond$s, add variable bindings after each quantifier, then check the formula on a path of length 2, where the first state represents falsehood and the second represents truth. Thus formula (2) would become

$$\Diamond\{x\}\Box\{y\}\Diamond\{z\}(t(z) \wedge t(x) \vee t(y))$$

where t is a predicate that is false when its argument refers to state 1 and true for state 2. This *would* work if $\Box$ meant "for all states" rather than "for all future states" and likewise for $\Diamond$. But since y cannot be a state previous to x and z cannot be previous to y, there are some assignments that will never arise from this path constraint, such as $\{x \mapsto 2, y \mapsto 2, z \mapsto 1\}$. Thus the path constraint is not equivalent to formula (2).

However, it is temptingly close, and it is easy to imagine language extensions that would make the proof work. For example, if we had an operator called @ that allowed us to jump to a previous state, we could translate formula (2) as

$$\{h\}\Diamond\{x\}@_0\Box\{y\}@_h\Diamond\{z\}(t(z) \wedge t(x) \vee t(y))$$

The @ operator exists in hybrid logic, and indeed precisely this approach was used by Franceschet & de Rijke in 2006 to prove that model-checking a hybrid logic with @ and variable binders is PSPACE-hard [88].

Proving that model-checking a path constraint is PSPACE-hard is more challenging. Instead of a simple two-state model, we must build a $2k$-state model, where k is the number of quantifiers. The odd states will represent falsehood; the even, truth. The following proof provides the details.

Theorem 2. *The problem of model-checking a path constraint on a single path is PSPACE-complete.*

Proof. By theorem 1, the problem is in PSPACE. To show it is PSPACE-hard, we exhibit a polynomial-time reduction of QBF. Let

$$Q_1 x_1 Q_2 x_2 \cdots Q_k x_k \phi(x_1, x_2, \ldots, x_k)$$

be a QBF in prenex normal form, so $Q_1, \ldots, Q_k$ are quantifiers and $\phi(x_1, \ldots, x_k)$ is an abbreviation for a propositional Boolean formula over the variables. We translate this QBF into a path constraint model-checking problem as follows. Define the signature by

$$\Sigma = ((x_1, \ldots, x_k), (\emptyset), (\{p_1, \ldots, p_k\}, \{t\}, \emptyset, \emptyset, \ldots))$$

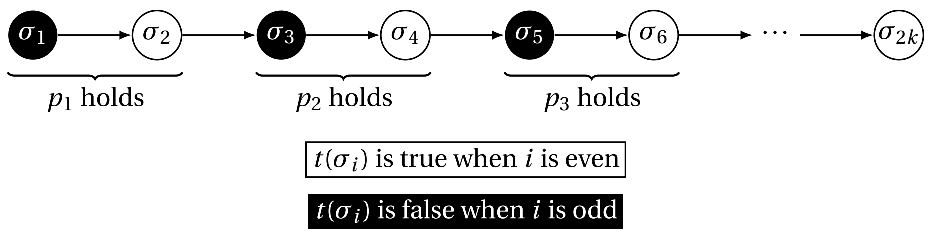


Figure 5. *A graphical representation of the path constraint language model that we construct in the reduction of a QBF.*

Thus $x_1, \ldots, x_k$ will be variables, $p_1, \ldots, p_k$ will be nullary predicates, and t will be a unary predicate. Now let σ be a temporal sequence of $2k$ states, $\sigma_1, \ldots, \sigma_{2k}$. We define the interpretation of the nullary predicates at each state by

$$\sigma_i(p_j) = \begin{cases} \textit{true} & \text{if } 2j-1 \le i \le 2j \\ \textit{false} & \text{otherwise} \end{cases}$$

for all i, j. (A nullary relation can be simply identified with *true* or *false*.) We define the interpretation of the unary predicate t by

$$\sigma_i(t) = \{\sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k}\}$$

Figure 5 summarizes this model. Note that we can construct this model in linear time.

We now construct the path constraint that corresponds to the QBF. For the quantifier part, we translate:

- $\forall x_i$ into the string "$\Box(p_i \to \{x_i\}$"
- $\exists x_i$ into the string "$\Diamond(p_i \land \{x_i\}$"

The quantifier-free part we transcribe literally, except that each occurrence of x_i becomes $t(x_i)$. At the end of the formula, we add k closing parentheses. For example, formula (2) becomes

$$\Diamond(p \land \{x\}\Box(q \to \{y\}\Diamond(r \land \{z\}(t(z) \land t(x) \lor t(y)))))$$

This formula, too, can be constructed in linear time. It is larger than the original QBF, but only by a constant factor.

It remains to show that the constructed path constraint (call it ψ) is true iff the original QBF (call it χ) is true. Let χ_i denote χ with the first i quantifiers removed. Similarly let ψ_i denote ψ with the first i quantifier translations (and the last i parentheses) removed. We will show that for any i, for any $j < 2i$, and for any assignment s of the variables,

$$\sigma, j, s \vDash_{\text{PCL}} \psi_i \quad \text{iff} \quad s^* \vDash_{\text{FOL}} \chi_i$$

(We use $\vDash_{\text{PCL}}$ and $\vDash_{\text{FOL}}$ to indicate semantic consequence with respect to our path constraint language and FOL, respectively.) Note that we translate the path constraint language assignment $s : V \to S$ into the FOL assignment $s^* : V \to \{\mathit{true}, \mathit{false}\}$ given by

$$s^*(x_j) = \begin{cases} \mathit{true} & \text{if } s(x_j) \text{ is one of } \sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k} \\ \mathit{false} & \text{if not} \end{cases}$$

The proof is by induction on i. The base case is $i = k$, where all the quantifiers have been removed, so χ_i and ψ_i are both propositional formulas. The propositional connectives have the same semantics in our path constraint language and FOL, so it suffices to show that for any $j < 2i$ and for each h,

$$\sigma, j, s \vDash_{\text{PCL}} t(x_h) \quad \text{iff} \quad s^* \vDash_{\text{FOL}} x_h$$

By the definition of t and the semantics of our path constraint language, it suffices to show that

$$s(x_h) \in \{\sigma_2, \sigma_4, \sigma_6, \ldots, \sigma_{2k}\} \quad \text{iff} \quad s^* \vDash_{\text{FOL}} x_h$$

This is immediate from the definition of s^*.

For the inductive step, suppose we have proven the result for $i+1$, so we know that for any $j < 2i+2$ and any assignment s, we have $\sigma, j, s \vDash_{\text{PCL}} \psi_{i+1}$ iff $s^* \vDash_{\text{FOL}} \chi_{i+1}$. We split by cases based on whether the ith quantifier is $\forall$ or $\exists$.

Case: $\forall$. We must show that for each $j < 2i$ and for any s,

$$\sigma, j, s \vDash_{\text{PCL}} \Box(p_i \to \{x_i\}\psi_{i+1}) \quad \text{iff} \quad s^* \vDash_{\text{FOL}} \forall x_i \chi_{i+1}$$

Whenever $j < 2i$, observe that

$$\begin{aligned} & \sigma, j, s \vDash_{\text{PCL}} \Box(p_i \to \{x_i\}\psi_{i+1}) \\ \text{iff} \quad & \sigma, h, s \vDash_{\text{PCL}} \{x_i\}\psi_{i+1} \text{ for all } h \geq j \text{ with } \sigma, h, s \vDash p_i \\ \text{iff} \quad & \sigma, h, s \vDash_{\text{PCL}} \{x_i\}\psi_{i+1} \text{ for all } h \in \{2i-1, 2i\} \\ \text{iff} \quad & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\text{PCL}} \psi_{i+1} \text{ for all } h \in \{2i-1, 2i\} \end{aligned}$$

Similarly,

$$\begin{aligned} & s^* \vDash_{\text{FOL}} \forall x_i \chi_{i+1} \\ \text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\text{FOL}} \chi_{i+1} \text{ for all } h \in \{\mathit{true}, \mathit{false}\} \end{aligned}$$

Thus, it suffices to show that

$$\begin{aligned} & \sigma, h, s[x_i \mapsto \sigma_h] \vDash_{\text{PCL}} \psi_{i+1} \text{ for all } h \in \{2i-1, 2i\} \\ \text{iff} \quad & s^*[x_i \mapsto h] \vDash_{\text{FOL}} \chi_{i+1} \text{ for all } h \in \{\mathit{true}, \mathit{false}\} \end{aligned}$$

For this, it suffices to show that

$$\begin{aligned} & \sigma, 2i-1, s[x_i \mapsto \sigma_{2i-1}] \vDash_{\text{PCL}} \psi_{i+1} \\ \text{iff} \quad & s^*[x_i \mapsto \mathit{false}] \vDash_{\text{FOL}} \chi_{i+1} \end{aligned}$$

and

$$\begin{aligned} & \sigma, 2i, s[x_i \mapsto \sigma_{2i}] \vDash_{\text{PCL}} \psi_{i+1} \\ \text{iff} \quad & s^*[x_i \mapsto \mathit{true}] \vDash_{\text{FOL}} \chi_{i+1} \end{aligned}$$

But each of these is simply an instance of the induction hypothesis, since by the definition of s^* we have

$$\begin{aligned} (s[x_i \mapsto \sigma_{2i-1}])^* &= s^*[x_i \mapsto \mathit{false}] \\ (s[x_i \mapsto \sigma_{2i}])^* &= s^*[x_i \mapsto \mathit{true}] \end{aligned}$$

Case: $\exists$. We must show that for each $j < 2i$ and for any s,

$$\sigma, j, s \models_{\text{PCL}} \Diamond(p_i \wedge \{x_i\}\psi_{i+1}) \quad \text{iff} \quad s^* \models_{\text{FOL}} \exists x_i \chi_{i+1}$$

By reasoning parallel to the $\forall$ case, it suffices to show that

$$\begin{aligned} &\sigma, h, s[x_i \mapsto \sigma_h] \models_{\text{PCL}} \psi_{i+1} \text{ for some } h \in \{2i-1, 2i\} \\ \text{iff} \quad & s^*[x_i \mapsto h] \models_{\text{FOL}} \chi_{i+1} \text{ for some } h \in \{\mathit{true}, \mathit{false}\} \end{aligned}$$

As in the previous case, this follows from the induction hypothesis.

This concludes the inductive proof that $\sigma, j, s \models_{\text{PCL}} \psi_i$ iff $s^* \models_{\text{FOL}} \chi_i$ for any i, any $j < 2i$, and any s. Choosing $i = 0$ and $j = 1$, we obtain $\sigma, 1 \models_{\text{PCL}} \psi$ iff $\models_{\text{FOL}} \chi$ as desired. □

The QBF reduction in this proof is similar to that recently employed to prove PSPACE-completeness for the finitary model-checking problem for LTL augmented with finitely many registers, over deterministic one-counter automata [72].

The PSPACE-completeness result here is somewhat unfortunate, but perhaps to be expected, since, as we have seen, the variable-binding extension is a sort of quantifier, and a great many interesting decision problems for quantified logic are PSPACE-complete [93], including even the problem of checking first-order monadic logic over finite paths [147]. And of course hybrid logic with $\downarrow$ has the same problem as our path constraint language.

There are practical reasons not to be overly concerned about the PSPACE-hardness of checking path constraints. First, actual path constraints are typically not terribly long. They are, after all, written by humans for the purpose of reasoning formally about constraints that arise naturally. Second, it is possible that the predicates that arise in this domain might lend themselves to specialized analyses with good performance. That is, the predicates we actually encounter are not arbitrary n-ary relations over the state space; they are simple tests like "has a database." Third, and most importantly, the constraints that arise are likely to be particularly well-behaved, so the worst-case running time will be quite rare in actuality. Of course, this is a claim that is likely to be made about any intractable problem, but at least here it can be formalized.

The complexity of the path-checking problem in our case arises, of course, from the variable-binding extension, but in particular it arises from *nesting* these binders. All of the intractable examples we have seen, such as the transformed formula in the QBF reduction, involve arbitrarily deeply nested variable bindings. The following theorem shows that model-checking a path constraint over a path is intractable only to the extent that variables are nested without bound.

Theorem 3. *Let σ be a temporal sequence of length m. Let ϕ be a path constraint of length ℓ. Let d be the maximum variable-nesting depth of ϕ (i.e., there is no point in ϕ at which more than d variables are bound at once). Then there is an algorithm that determines whether $\sigma, i \models \phi$ in $O(\ell m^{d+1})$.*

Proof. Let $\phi_1, \ldots, \phi_n$ be the subformulas of ϕ, listed by nondecreasing length. We define a Boolean matrix $[t_{j,k}]_{m \times n}$. The follow recursive algorithm fills the kth column of the matrix under variable assignment s:

```
FillCol($k, s$)
    if $\phi_k$ is an atomic proposition
        for $j = 1$ to $m$
            $t_{j,k}$ := evaluate $\phi_k$ at $j$ using assignment $s$
    if $\phi_k = false$
        for $j = 1$ to $m$
            $t_{j,k} := false$
    if $\phi_k$ is of the form $\phi_g \rightarrow \phi_h$
        FillCol($g, s$)
        FillCol($h, s$)
        for $j = 1$ to $m$
            $t_{j,k} := t_{j,h}$ or not $t_{j,g}$
    if $\phi_k$ is of the form $\bigcirc\phi_h$
        FillCol($h, s$)
        for $j = 1$ to $m - 1$
            $t_{j,k} := t_{j+1,h}$
        $t_{m,k} := true$
    if $\phi_k$ is of the form $\phi_g$ U $\phi_h$
        FillCol($g, s$)
        FillCol($h, s$)
        $prev := false$
        for $j = m$ to $1$ step $-1$
            $prev := t_{j,h}$ or ($t_{j,g}$ and $prev$)
            $t_{j,k} := prev$
    if $\phi_k$ is of the form $\{x\}\phi_h$
        for $j = 1$ to $m$
            FillCol($h, s[x \rightarrow \sigma_j]$)
            $t_{j,k} := t_{j,h}$
```

To solve the model-checking problem, execute FillCol($n, \emptyset$) and read the answer from $t_{i,n}$. The correctness of the algorithm follows easily from the semantics of path constraints.

Note that in the case where there are no variable assignments, this algorithm reduces to the algorithm for LTL. When there are variable assignments, intermediate results in the matrix are overwritten; in an intermediate evaluation of a formula of the form $\{x\}\psi$, we use the columns to our left to first evaluate ψ under the assignment $x \mapsto 1$, then immediately overwrite them, using the same space to evaluate ψ under the assignment $x \mapsto 2$, and so on. More precisely, the kth column of the matrix is ultimately filled m^{d_k} times, where d_k is the variable-nesting depth of ϕ_k (i.e., the number of variables that are bound for the subformula ϕ_k). Thus no column is filled more than m^d times, where d is the maximum variable-nesting depth of ϕ. Filling a single column, not counting the time to fill its subformula columns to the left, takes $O(m)$ time. And there are n columns in total, which is $O(\ell)$. Therefore the total complexity of the algorithm is $O(\ell m^{d+1})$.

□

If d is bounded (e.g., we never have constraints with a variable-nesting depth greater than 3) then we can model-check a path constraint over a path in polynomial time. (If $d = 0$, we get the same, linear performance as the LTL algorithm, as we would hope.) If d is unbounded, then the performance is exponential, $O(\ell m^{\ell})$, since the quantifier depth can approach the length of the formula. In practice, it seems unlikely that variable bindings will often be very deeply nested, since software architects are unlikely to be naturally interested in such convoluted constraints. The path constraints that arose in our empirical work (chapter 5) seem to bear this out; in fact, I have never seen a path constraint arising from a real-world evolution with a variable-nesting depth greater than 1.

3.5 Summary

In this chapter, we produced a formal syntax and semantics for the path constraint language that was introduced in section 2.4, surveyed a number of related logics, and proved some results about the computational complexity of verifying evolution path constraints. This chapter thus addresses research claim 2 from section 1.3.3, which was a claim about the computational complexity of path constraint verification.

In one sense, this kind of formal analysis provides the best and most thorough possible evaluation of the research claim. A result of PSPACE-completeness establishes a tight upper and lower bound on both the time complexity and space complexity of verifying path constraints. This is the best sort of result we could hope for in undertaking a theoretical evaluation like this one.

In another sense, though, the practical relevance of these results is somewhat murky. The PSPACE-completeness result (theorem 2) suggests that verifying path constraints may be intractable for large problem sizes, and the polynomial-time result for the case in which the variable-nesting depth is bounded (theorem 3) suggests that the problem may become tractable provided that rigid variables are not deeply nested. But how are we to apply these theoretical results in practical contexts? Is path constraint verification computationally practical for the problem sizes and constraint specifications that are likely to emerge in realistic architecture evolution scenarios?

This dissertation does not provide final answers to these questions. Fully settling them would require performance tests of tools implementing our approach. Thus, although these theoretical results provide a final and decisive answer to the question of the *computational complexity* of path constraint verification, they do not provide an entirely satisfying answer to the question of *practical performance*. I leave this for future work. (However, see section 6.3 for a discussion of an implementation project that incorporated automated enforcement of path constraints, demonstrating that evolution paths satisfying a reasonable set of realistic path constraints can be automatically generated fairly quickly.)

It is also worth emphasizing that these theoretical results are of much broader relevance beyond their immediate application to the topic of software architecture evolution. As we saw in section 3.3, this version of linear temporal logic augmented with state-capturing variables has been reinvented numerous times and applied to a variety of practical contexts. The theoretical results proved in this chapter are of relevance to anyone using such a logic to verify formulas over finite paths. This finite-path computational complexity question, although it might naively be thought to be more trivial or obvious than the problem of model-checking a temporal formula over an entire Kripke structure, is of substantial theoretical interest and practical importance [147]. Thus, it is my hope that the results here will be useful to other researchers beyond the domain of architecture evolution, apart from their relevance to my own thesis.

4 Case study: Architecture evolution at NASA's Jet Propulsion Laboratory

In this chapter and the one that follows, I review the empirical research we have carried out to evaluate the applicability of our approach to architecture evolution in real-world software organizations. In this chapter, I discuss the first of the two case studies, which examined software architecture evolution at NASA's Jet Propulsion Laboratory.[6] This case study had three main goals: (1) to understand a real-world software architecture evolution problem in its natural context, (2) to assess the usefulness of our framework for software architecture evolution in helping to plan evolutions and reason about trade-offs, and (3) to assess the ease of implementing our approach to software architecture evolution with off-the-shelf languages and tools.

This chapter is organized as follows. Section 4.1 provides some background on software architecture evolution at JPL. Section 4.2 describes the evolution that was modeled in this case study. Section 4.3 describes how I modeled this evolution, adapting our architecture evolution modeling approach to architecture modeling languages and tools in use at JPL. Section 4.4 present the results of this modeling process. Finally, section 4.5 concludes by discussing the significance of these results.

4.1 Background

At JPL, software architecture evolution is both very common and very important. One reason for this is that missions at JPL can last for decades, and software systems must evolve to support them continuously. The Voyager mission launched in 1977, and 36 years later it is still running—and transmitting telemetry that must continue to be processed by software on the ground. The flight software and ground software associated with this mission have required continuous maintenance to keep them up and running for 36 continuous years. This maintenance entails not only routine collection and analysis of telemetry, but also occasional software evolution as well as responses to sporadic anomalies, as in 2010 when a flight software glitch left Voyager 2 nonfunctional until project engineers could repair the software, allowing it to continue reporting on its journey out of our solar system [56]. The Voyager probes

[6]The case study described in this chapter was carried out during an internship at JPL in the summer of 2011. The case study report from my internship first appeared publicly on the JPL Technical Report Server [15]. In 2012, I published a conference paper on the case study [16].

I would like to express my deep gratitude to Brian Giovannoni, Dave Santo, and Oleg Sindiy for their guidance and support on this project.

are expected to continue transmitting telemetry at least until 2025, when they will at last have insufficient power to support any of their instruments, for a total mission length of nearly half a century [113].

But long mission durations are only one reason that software evolution is so important to JPL. Perhaps even more significant are the *multimission* software systems that JPL maintains. Today, JPL is constantly maintaining the software for a wide array of missions; currently JPL has 119 active missions, according to its website—from the Active Cavity Irradiance Monitor Satellite to the Wide-Field Infrared Survey Explorer [115]. Each of these missions has plenty of custom software written for it, but most of them also make use of multimission software—software that is shared among several missions. JPL takes a sort of product line approach to multimission software; it develops software for multimission use, then adapts it for each mission. Of course, multimission software lasts longer than a typical single mission, and it also has greater evolution needs. As new missions make use of a multimission platform, the platform must evolve to support the new capabilities and qualities that the new missions require. Over a long period of time, a multimission system can change drastically, ultimately to the point where it bears little resemblance to its ancestral form.

The best example of such a multimission system at JPL is the Advanced Multimission Operations System. AMMOS is the ground software system used for JPL's deep-space and astrophysics missions [114]. It was developed beginning in 1985, with the goal of providing a common platform to allow mission operators to manage ground systems at lower cost than would be possible by building mission-specific tools, without compromising reliability or performance [103]. The system has been used for many prominent NASA missions, and continues to be used today.

Architecturally, AMMOS is a system of systems; although it functions as a coherent whole with a common purpose, it is composed of disparate elements, each of which has its own engineers, its own users, and its own architectural style. Among the systems that make up AMMOS are elements responsible for uplink and downlink of spacecraft telemetry, for planning command sequences, for processing spacecraft telemetry, for navigation, and so on.

AMMOS has served JPL well for many missions, but it is an aging system, and by the time of this case study, the limitations of its architecture had become apparent [163]. The architecture was resistant to evolution and expensive to maintain. The system suffered from architectural inconsistencies and redundancies and lacked a coherent overarching architecture. Requirements changes often necessitated modifications spanning many subsystems, and the system relied on large amounts of "glue" code—adapters and bridges connecting different parts of the system in an ad hoc way that made maintenance difficult.

At the time of this case study, ongoing architecture modernization efforts aimed to address this situation by rearchitecting AMMOS in a way that would make use of modern architectural styles and patterns [163; 194]. This would allow easier, less expensive maintenance and evolution of AMMOS in the future and also facilitate easier customization of AMMOS for individual missions. The goal was to develop a modern deep-space information systems architecture based on principles of composability, interoperability, and architectural consistency.

4.2 Evolution description

For the case study, I focused on one element of AMMOS that was of particular interest from the standpoint of software architecture evolution: the AMMOS element responsible for mission control, data management and accountability, and spacecraft analysis (MDAS). The MDAS element was an attractive choice for several reasons. First, it was undergoing a major restructuring to meet specific goals. Second, it had an explicit initial software architecture, and the target architecture was also reasonably well understood. Third, it presented interesting trade-offs and unanswered questions that might be usefully addressed by an architecture evolution analysis. Fourth, I had good access to staff who were familiar with the system and with the evolution, who could provide architectural information beyond that available in official documentation.

The MDAS element has a number of responsibilities, but one of the most important is to process, store, and display telemetry and other mission data from deep-space operations. Prior to the Mars Science Laboratory (MSL) mission, this responsibility was fulfilled by an assortment of different subsystems, including the Data Monitor and Display (DMD) assembly; the Tracking, Telemetry, and Command (TT&C) system; and a number of others [124]. For the MSL mission, a new system was developed to supplant this complex of systems: the Mission Data Processing and Control System (MPCS) [69].

MPCS was originally developed as a testing platform modeled after the ground data systems for the Mars Exploration Rovers; later it was promoted to support operations for MSL [69]. At the time of this case study, engineers were in the process of adapting and refining the architecture of MPCS for multimission use. In this section, I describe the initial MPCS architecture (as used by MSL), the motivations for evolving it, and the planned future architecture.

4.2.1 Initial architecture

MPCS was architected as an event-driven message bus system, with all communication among the system's major components occurring via a Java Message Service message bus [69]. This architecture was designed to promote loose coupling of software components without compromising reliability. Components could be attached to or detached from the message bus freely (by subscribing to or publishing the appropriate kind of event) provided that they adhered to application protocols and did not violate architectural constraints, allowing for plug-and-play reconfiguration of the system. The components were Java-based and platform-independent; the interfaces by which they communicated were based on XML [69].

This event-driven, bus-mediated architecture gave MPCS a degree of architectural flexibility. There was not really any one "MPCS architecture;" rather, MPCS could be configured in different ways to achieve different goals. At its most flexible, MPCS could be regarded a loose confederation of tools rather than a cohesive system with a fixed design. However, MPCS did have a rather stable infrastructure of core components that were generally connected in a well-defined way, so for most purposes MPCS could be treated as a system with a stable platform and a fixed set of variation points.

An important example of the architectural variability of MPCS is that it could be deployed with different configurations in different environments. MPCS was used in several environments—not only mission operations, but also flight software development; system integration; and assembly, test, and launch operations (ATLO)—and there were initially significant differences in architectural configuration among the environments [69]. For flight software development, for example, MPCS could be used to issue commands to the flight software under development; in operations, however, commanding features were delegated to a different system called CMD, which was external to MPCS and indeed external to AMMOS (it was a subsystem within JPL's Deep Space Network).

The most important components of MPCS are:

- The aforementioned **message bus**.
- The telemetry processing subsystem of MPCS, called **chill_down** [45] (*chill* is a code name for MPCS [212], and *down* is for *downlink*). The role of this component is to take as input an unprocessed telemetry stream from a spacecraft (or other telemetry source, such as a simulation environment), perform frame synchronization and packet extraction, and process packets to produce event verification records and data products [45].
- The commanding component of MPCS, called **chill_up** (*up* for *uplink*) [67; 68]. This component's role is to transmit commands to the flight software (or simulation environment). In the initial architecture, chill_up was used only in the flight software development and ATLO environments, not in operations.
- The **MPCS database**, a MySQL database used for storing telemetry as well as some other information, such as logs and commanding data [68]. This database was queried by a number of analysis components.
- The monitoring interface, called **chill_monitor**, used for real-time display of telemetry [67; 68]. There were generally many instances of chill_monitor for a single instance of MPCS, as many mission operators could be monitoring telemetry simultaneously.
- A variety of **MPCS query** components, with names like *chill_get_frames*, *chill_get_packets*, *chill_get_products*, and so on [68]. The purpose of these programs was to retrieve data from the database and output the data in a standard format.

Together, these components effectively formed a standard MPCS workflow. Commands would be issued by chill_up and conveyed to the flight software (or simulation environment), which would carry out the commands; the flight software would produce telemetry, which would be processed by chill_down. The chill_down component would store the processed telemetry to the MPCS database (where it would be queried by the MPCS query components) and transmit messages about the processed telemetry to the message bus (where it would be displayed by chill_monitor). Although MPCS was flexible enough to be configured in many different ways, this workflow describes the way MPCS works most of the time, in typical environments. In section 4.4, I will show an architectural diagram of MPCS illustrating these components and the connections among them.

4.2.2 Impetus for evolution

MPCS was expected to serve adequately for the MSL mission. However, as MPCS was to be developed for reuse in future missions, engineers faced the need to evolve the system to improve qualities such as performance and usability, support additional needed capabilities, and better integrate with other ground data systems. In this case study, I focused on two particular proposed features of MPCS that project architects hoped to introduce in future versions: *integrated commanding (ICMD)* and *timeline integration*.

ICMD was motivated by the NASA principle "test like you fly." NASA aims to make system-testing environments as similar as possible to actual spaceflight operations. As we have seen, one of the most salient architectural characteristics of MPCS in the initial architecture is that it took different forms in different environments. In particular, there were important architectural differences between the testing environment (ATLO) and the spaceflight operations environment. The ICMD effort aimed to bring the operations environment more in line with the ATLO environment.

The main difference between the testing configuration of MPCS and the operations configuration of MPCS was commanding. In ATLO, the chill_up component of MPCS was responsible for issuing commands to the spacecraft. In operations, the responsibility of issuing commands was excised from MPCS entirely; instead, the CMD system was responsible for issuing commands. ICMD was intended to change the operations environment to look more like ATLO; the chill_up component of MPCS would now issue commands in all environments.

Timelines were a new data structure proposed for storing streams of time-oriented data throughout AMMOS. A "timeline" is exactly that: a linear sequence of events with associated times, in chronological order. Many of the kinds of data that JPL handles on a day-to-day basis fit naturally into this model: telemetry, command sequences, and others. The timeline proposal defined specific formats for storage and transmission of timelines, and also described the architectural infrastructure necessary to support them. Timelines were expected to be useful for many purposes, but one of the most important was comparison of actual telemetry with expected telemetry. Mission operators need this capability all the time (comparing an observation with a theoretical prediction is one of the most basic requirements in science), but in the initial architecture, comparing expected and actual telemetry was a manual, laborious operation. Supporting timelines would require substantial architectural infrastructure. Although the basic idea is not complex, there were very stringent performance requirements; processing timelines had to be very fast. Thus, for example, a specially engineered timeline database was planned, which would be designed specifically for efficient storage and retrieval of timelines.

The introduction of timelines would have ramifications for many of the AMMOS elements, including MPCS. In its initial form, MPCS stored telemetry information in a MySQL database. Ultimately, this database was to be rendered obsolete by the introduction of timelines. After timelines are integrated into MPCS, telemetry would be stored in an AMMOS-wide timeline management system, and the MySQL database would eventually be retired. Other parts of MPCS would also be affected by the introduction of timelines; for example, the subsystem for mission planning and sequencing is likely to see changes as well.

4.2.3 Target architecture

The main differences between the end state of the evolution and the initial state were greater homogeneity among deployment environments (supporting the "test like you fly" philosophy) and integration with the new timeline infrastructure (improving on the usability of the existing architecture).

In the end state, chill_up would be used for commanding in all environments, including spaceflight operations. The CMD system would continue to exist but would no longer be the originator of commands in the operations environment. Instead chill_up would originate commands and convey them to CMD, which would prepare them for uplink.

More precisely, chill_up would not directly send commands to CMD, but would instead transmit references to commands that it had stored in a command repository; CMD would then access this repository to read the actual commands. With the introduction of timelines, this command repository would become obsolete, as command sequences would be stored as timelines and would therefore be stored by the timeline management system. This was an important point of interaction between the two pieces of the evolution, ICMD and timelines.

In addition to storing commands, the timeline management system in the end state would be responsible for storing channelized telemetry. Thus, the target architecture lacked the MySQL database that existed in the initial architecture, and the usages of that database by other MPCS components were replaced by connections with the timeline management system. The timeline management system would be external to MPCS, so these connections would be external collaborations rather than internal connections.

4.3 Approach

At the beginning of the case study, I spent several weeks gathering information—familiarizing myself with the particulars of the various elements of AMMOS and the plans that were in place for evolving them. I did this by reviewing project documents and speaking with project personnel.

During this period, I also selected an evolution to study—the one described in the previous section. Among the options considered were past evolutions (i.e., those that had already been finished and whose outcome was therefore known); current evolutions (i.e., those which were ongoing); and future evolutions (i.e., those that were under consideration to occur in the future). I ultimately selected an evolution that was in progress but had been underway for only a short time. Picking a current evolution had the advantage of being of greatest relevance to JPL. Another advantage was that there were ample resources for learning about the evolution; it was easy to find project personnel who could share accurate, timely information about evolution plans. If I had selected a past evolution, it is likely that documentation would have been difficult to find, and there would have been few available personnel who were familiar enough with the evolution to provide useful information. On the other hand, if I had selected a future evolution for which few firm plans had been made, substantial speculation would have been necessary in order to construct an evolution graph.

Another important choice was the scope of the evolution, in terms of both time (i.e., a long evolution versus a short one) and breadth (i.e., the evolution of a small subsystem versus the evolution of a large chunk of AMMOS). I could have picked a much larger scope than I did—for example, by studying the overall evolution of AMMOS rather than focusing on MPCS, or by trying to look further into the future. However, given the limitations of the case study format, it would have been difficult to gather sufficient information about a broader evolution to produce a useful model capable of saying anything useful about the evolution—one that was more than a superficial overview. A more narrowly scoped study, on the other hand, would have shown changes that were too minor to be interesting.

Once I had selected an evolution to study, the next task was to model it. As mentioned in the introduction to this chapter, one of the aims of this work was to evaluate the practicality of adapting our approach to off-the-shelf languages and tools like those used at JPL. At JPL, the dominant modeling language is SysML, the Systems Modeling Language, and the dominant modeling tool is MagicDraw, a commercial tool that can produce SysML models. In this section, I will describe how I adapted our architecture evolution approach to SysML and MagicDraw.

4.3.1 Representing software architecture in SysML

SysML [173] is a specialization of UML to the domain of systems engineering. It is defined as a *profile* of UML. SysML arose from collaboration, beginning in 2001, between the Object Management Group and the International Council on Systems Engineering. It was developed by a coalition of industry leaders and adopted as a standard in 2006. SysML is both a restriction and extension of UML. It is an extension in the sense that it adds new syntax and semantics beyond that of UML. It is a restriction in that it excludes many of the elements that do exist in UML, for the purpose of simplifying the language. SysML takes a subset of the diagram types from UML and repurposes them for the domain of systems engineering. The class diagrams of UML, for example, become block definition diagrams (BDDs) in SysML; composite structure diagrams become internal block diagrams (IBDs).

I used two diagram types in representing evolution states: BDDs and IBDs. In SysML, a *block* is the basic unit of system structure. A BDD shows the blocks that appear in the model; an IBD shows the internal structure of a block. I used BDDs to show the kinds of architectural components in my model and the hierarchical relationships among them; I used IBDs essentially as conventional software-architectural diagrams, to show the architectural structure (components, connectors, etc.) of a system. BDDs and IBDs are both representations of an underlying model.

I tailored my use of the diagram types to show those aspects of the architecture whose evolution I hoped to model. The most detailed and important diagram that I produced was an IBD showing the internal structure of MPCS. I also produced a set of three IBDs that served essentially as context diagrams showing how MPCS was deployed in the three different environments (flight software development, ATLO, and operations). For representing this evolution, it was important to see not only the internal changes that were occurring within MPCS, but also the changes in how MPCS interacted with other systems, such as CMD. Recall that such changes were key to the overall evolution, so modeling them was crucial to providing a complete, useful representation of the evolution.

4.3.2 Modeling software architecture evolution with MagicDraw

Modeling software architecture in SysML is fairly straightforward; after all, SysML is a profile of UML, which is specifically designed for software architecture representation. More interesting is the question of how to model architecture *evolution* effectively. Recall that we model an evolution as a graph, in which the nodes are intermediate architectures and the edges are transitions. The first step in representing an architecture evolution, then, is figuring out how to represent the nodes. The simplest strategy would be to create one MagicDraw project for each intermediate state. A better idea, however, is to include all the intermediate states, and hence the entire evolution graph, in a single project. With everything in one project, it becomes possible to write evolution constraints and evaluation functions with existing tools, simply by using the model constraint and analysis facilities already provided by the tool.

More specifically, I placed each intermediate state in its own package. A package is a UML construct (also available in SysML) that encapsulates related entities. Placing the intermediate states into different packages allows them to be as isolated as necessary, while still existing within the same project so as to accommodate analyses of the entire evolution graph. In addition, we can represent the packages themselves in a *package diagram*; then we can represent the transitions between them by relationships among the packages. Finally, if we wish, we can add annotate these packages and relationships with additional information to facilitate analysis, such as node properties and edge properties.

4.3.3 Representing model transformations

The problem with having a separate representation for each intermediate state is that it can be a maintenance nightmare. Since what I was modeling was a gradual evolution rather than an outright retirement and replacement of a system, all the states look mostly the same, except for those pieces that are evolving. Thus, modeling the evolution graph required me to produce many nearly identical packages. I could have created this evolution graph model very easily by simply cloning the initial state and modifying it. That is, after first representing the initial state, I could have copied it, pasted it, and modified it to create the next state; then done likewise for the next state; and so on. But maintaining this evolution graph would have been painful.

Suppose that after I had finished representing all the states, I had noticed a mistake in the initial architecture that affected all the other states as well (since they were generated by cloning the initial state). To address the problem, I would have had to fix each state by hand. In the evolution graph I ultimately produced, there were seven states; a more broadly scoped evolution could have many more. Thus, fixing problems in this way would be laborious.

Instead of this copy-and-paste approach, I decided to model the structural transformations themselves in such a way that they could be applied automatically. Rather than generating intermediate states by hand and applying the evolution steps by hand, I would specify the structural transformations needed to generate the intermediate states automatically. Then, if the initial state were to change, the intermediate states could be regenerated instantly, so fixes could be applied in one place instead of many. This is analogous to the way that most revision control systems use delta encoding to store file versions (storing diffs between versions rather than a complete copy of every version of every file) or to the way that video compression works (by storing differences between frames, taking advantage of the typical similarity of nearby frames, rather than storing a complete copy of every frame).

This approach accords well with our approach to architecture evolution modeling. In our appraoch, we define evolution operators to capture the structural transformations involved in evolution steps, as well as other information to support analysis. The transformations that I employed in this case study fulfilled the same role here (except without providing metadata to support analysis).

I implemented the transformation specifications as macros in MagicDraw. MagicDraw supports the definition of fairly sophisticated macros that can alter both the model and the presentation of its diagrams. To do so, it exposes a rich Java API for creating and modifying model elements and presentation elements. Macros are written in a scripting language and compiled to Java bytecode.

There are several languages in which MagicDraw macros can be written: Groovy, BeanShell, JavaScript, JRuby, and Jython. All of these are dynamically typed programming languages that can compile to Java bytecode and run on the Java platform, so the choice among them is largely one of personal preference. I selected Groovy, whose syntax is based on Java but is rather laxer (e.g., semicolons and type declarations are unnecessary) and also introduces many additional features (e.g., for functional programming).

In principle, one could use a UML transformation standard such as QVT for this purpose, rather than a script using a proprietary API like MagicDraw's. I used macros rather than QVT for two reasons. First, MagicDraw had no built-in support for QVT (nor any other model transformation language), and although there was an official MagicDraw plug-in for QVT, it was immature, somewhat buggy, and not well documented. Second, using macros allowed me to transform not only the model, but also the diagrammatic presentation of that model. With QVT I would have been limited to the former; I could have transformed the model automatically, but still would have had to update the diagrams by hand, eliminating much of the benefit of the automated approach.

4.3.4 Modeling constraints and evaluation functions with OCL

Constraints and evaluation functions are important parts of our model of software architecture evolution. Due to time limitations, I was not able to incorporate formalization of constraints and evaluation functions as part of this case study. However, it is still useful to point out how constraints and evaluation functions could be formalized in principle in a model such as this one. With the entire evolution graph represented in a single model, constraints and evaluation functions could be represented as OCL constraints over the model.

OCL, the Object Constraint Language, is a declarative language for specifying rules of models in UML (and other modeling languages governed by the same metamodel as UML) [172]. OCL was originally developed to annotate UML models with additional constraints that are inexpressible in UML; however, it can also be used to express constraints over UML models—to judge whether a particular UML model satisfies some constraint. This makes it suitable for expressing evolution constraints, given that our entire evolution graph is in one model.

In principle, I believe that it should even be possible to develop an algorithm for translating, or compiling, constraints in our temporal-logic-based constraint language into OCL. Thus it would be possible to develop, say, a MagicDraw plug-in that allows architects to express constraints in this temporal logic, then transparently compiles them to OCL and checks them against the model. However, I have not explored this idea, and I leave the issue of constraint compilation for future work. (See also a similar discussion on compiling constraints into PDDL in section 6.3.2.)

Of course, macros are an option here too, and might provide some additional flexibility, at the cost of portability. Macros might be particularly useful for expressing evaluation functions, as OCL's constraint-based approach may be too rigid for quantitative analysis of the evolution graph.

4.4 Results

4.4.1 Representing the initial architecture

Figures 6 and 7 show the most important diagrams from my model of the initial architecture of MPCS. Figure 6 is an IBD depicting the internal structure of MPCS. This is a fairly complicated diagram, but there are a couple of features that are particularly worthy of attention. Note first the major components I mentioned in section 4.2: the message bus, the chill_up and chill_down (uplink and downlink) components, and so on. This is a very data-flow-oriented representation of MPCS, which is appropriate given its nature. Most previous architectural representations of MPCS at JPL have also depicted data flow prominently (e.g., [67–69]).

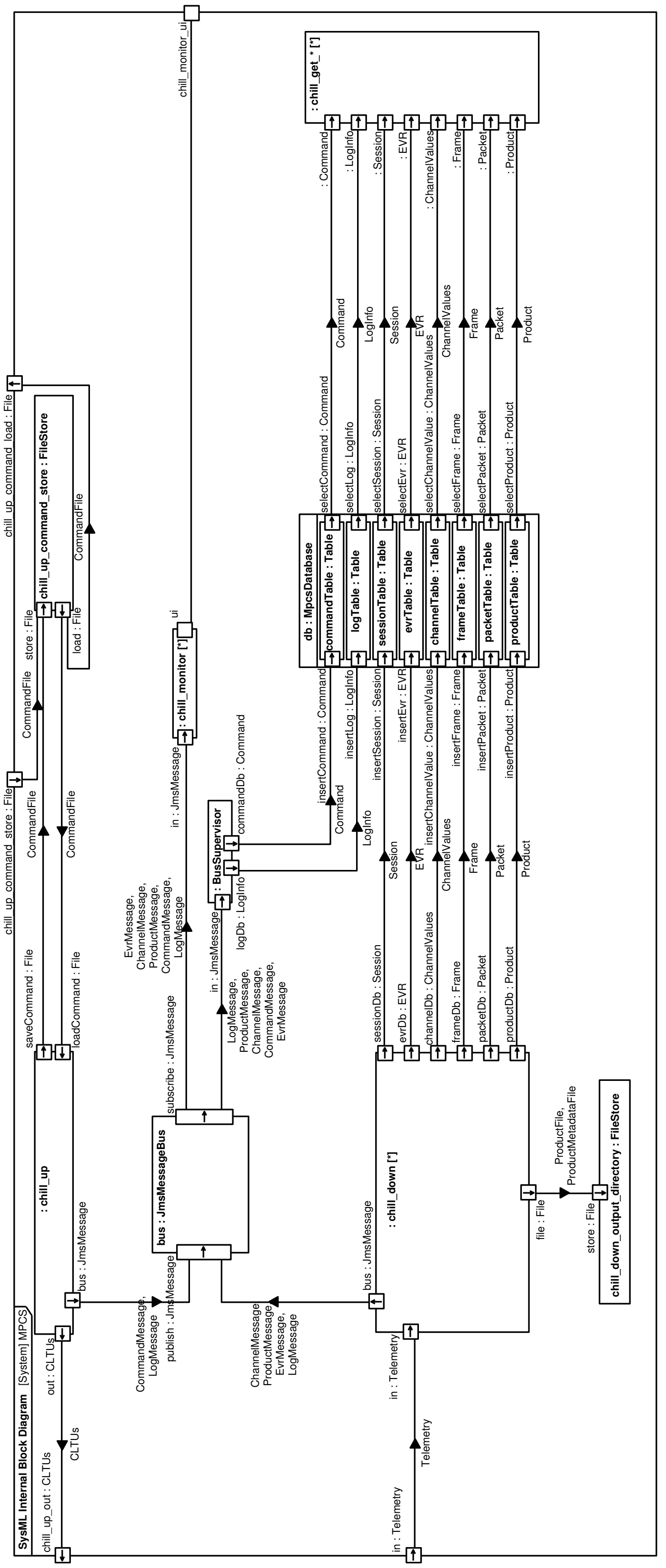


Figure 6. *IBD showing the internal structure of MPCS in the initial state.*

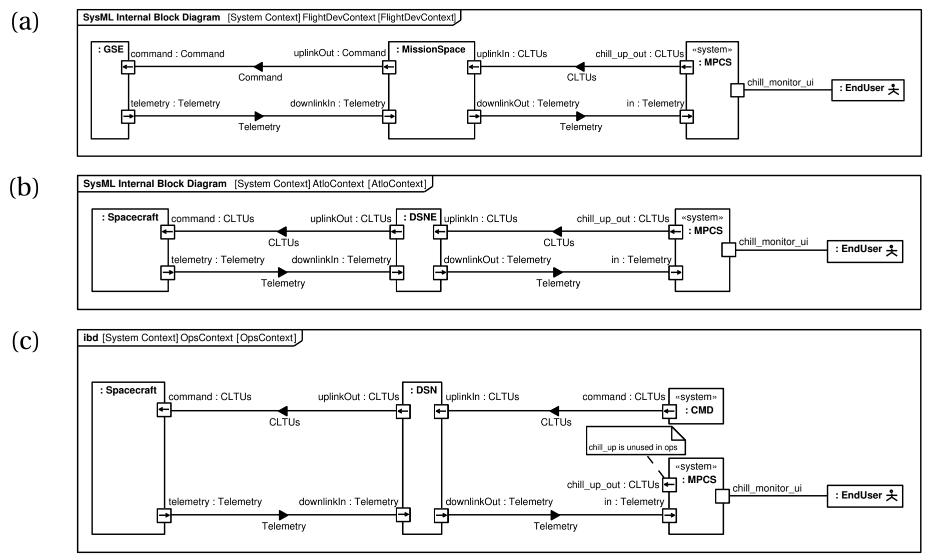


Figure 7. *IBDs showing the operational contexts in which MPCS can be deployed: (a) flight development, (b) ATLO, and (c) spaceflight operations.*

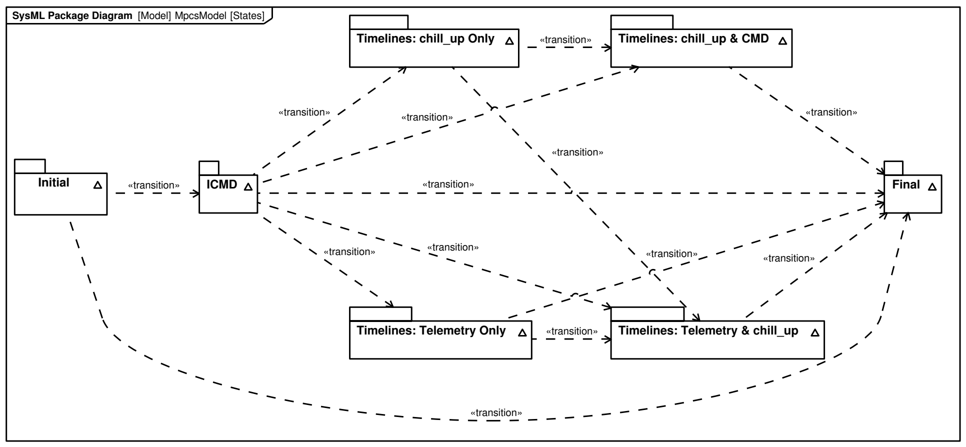


Figure 8. *Package diagram showing the evolution graph; states are represented as packages, and transitions are represented as dependencies (dashed lines).*

In other ways, however, this representation is quite different from previous representations of MPCS at JPL. The most important difference is that there are some key software elements that previous representations have depicted as being components of MPCS, but that I have represented instead as external collaborators of MPCS. To put it another way, I have drawn the boundary of MPCS differently from others at JPL who have represented the system. Previous representations have included inside MPCS components such as MissionSpace, a flight software development simulation environment. I have instead represented MissionSpace, along with all other environment-specific components, as external collaborators of MPCS. This has two advantages. First, it makes it easy to depict MPCS without the difficulty of somehow representing all the different architectural configurations in which MPCS can be deployed. Previous diagrams of MPCS have either addressed this issue by introducing special notation to indicate MPCS components that exist in some environments but not others, or ignored it by tacitly representing only one environment. The second advantage is that these extra components are not really part of MPCS anyway. MissionSpace, for example, is a third-party off-the-shelf tool, and no one thinks of it as being a component of MPCS; previous diagrams have included it as an MPCS component apparently for the sake of convenience and diagrammatic simplicity.

Of course, redrawing the boundary of MPCS in this way does not eliminate the problem of representing multiple environments; it merely pushes the problem outward, so we can deal with it separately. We still do need to represent the different environments, because they feature importantly in the evolution under study (the different architectural configurations evolve differently). Therefore, I produced three more IBDs that show how MPCS interact with its external collaborators (figure 7). Informally, I refer to these as *context diagrams*. Properly, neither UML nor SysML has a context diagram type, so I represented them as IBDs—partial internal representations of the larger system of systems in which MPCS resides (*partial* because I do not include all the ground data system elements, but only the small part of the system that is relevant to MPCS).

These context diagrams show the different environment configurations in a simple way. In subfigure 7a, we see that in the flight software development environment, MPCS both issues commands to and processes telemetry from the simulation equipment. Subfigure 7b shows the ATLO environment, which is mostly the same except that now MPCS is talking to the real spacecraft instead of a simulator. Finally, in subfigure 7c we see the spaceflight operations context, which is different. Here, a separate system is now responsible for commanding, while the uplink port on MPCS is unused.

4.4.2 Intermediate states and alternative paths

I ultimately produced an evolution graph with seven states, including the start and end states. The package diagram in figure 8 shows these states. The mainline evolution path is the simple, two-transition path from the “Initial” state to the “ICMD” state to the “Final” state. The first transition is the introduction of ICMD, and the second is the introduction of timelines. However, a number of alternative paths are possible.

The simplest possible path is to go directly from the initial state to the target state, skipping the ICMD evolution entirely. That is, rather than first integrating MPCS commanding into the spaceflight operations and then integrating timelines, we could go straight to the target architecture. This makes sense because the ICMD and timeline evolutions interact, and in some respects the timeline evolution undoes part of the ICMD evolution. The ICMD evolution rewires the commanding components of MPCS so that they communicate with the CMD element of the Deep Space Network; the timeline evolution then rewires these same components again so that they can communicate with the timeline management system. As is often the case with evolution paths, there are trade-offs. Going directly to the target state would be faster and cheaper than going via the ICMD waypoint. However, it would also be riskier—not only because the lack of intermediate releases increases the engineering risk, but also because the lack of stakeholder visibility into the state of the system would increase the risk of project cancellation.

The other alternative paths in this graph emerge during the introduction of timelines. One possibility would be to stage the introduction of timelines instead of introducing them all at once. In particular, the integration of timelines into the uplink portion of MPCS and the integration of timelines into the downlink portion are independent and could be accomplished separately.

Another evolution option has to do with the way that the chill_up component of MPCS interacts with CMD. In the final state, chill_up does not actually send commands directly to CMD; instead, it stores commands to the timeline management system, then passes a references to the command timeline to the commanding element. Instead of integrating timelines into chill_up in this way immediately, we could introduce an intermediate state in which chill_up makes use of the timeline management system itself but continues to send commands to CMD directly.

All of these various possibilities, and the complex interactions between them, appear in figure 8. Behind each of the packages in figure 8 is a complete architectural representation of the system in that state; here, I have shown only one state, the initial state (figures 6 and 7). In the next subsection, I describe how these intermediate-state representations are generated.

4.4.3 Representing architectural transformations

As I said in section 4.3.3, I used macros to specify the architectural transformations that defined the evolution transitions rather than explicitly specifying each intermediate state by hand. In addition, I limited the size of the transformation macro by building up the transformations out of smaller, reusable pieces. For example, in figure 3, the transition from "ICMD" to "Timelines: chill_up Only" and that from "ICMD" to "Timelines: Telemetry Only" both involve the introduction of a timeline management system, so rather than specify the introduction of the timeline management system twice, I defined it in such a way that it could be referenced by both transitions.

The transformation specification for the entire evolution graph was 752 lines of Groovy code, including transformations of both the model and the presentation of all diagrams (and excluding blank lines and comments). The code is reasonably easy to read and write. For example, here is the code that creates the timeline management system block (and its ports):

```
// Create new TMS block in the model
shared.tms = factory.createClassInstance()
shared.tms.name = "TMS"
shared.tms.owner = modelRoot
StereotypesHelper.addStereotypeByString shared.tms, "System"

// Add input port to TMS
shared.tmsInPort = factory.createPortInstance()
shared.tmsInPort.owner = shared.tms
shared.tmsInPort.name = "in"
StereotypesHelper.addStereotypeByString shared.tmsInPort, "FlowPort"
SysMLHelper.setDirectionFlowPort shared.tmsInPort, "in"

// Add output port to TMS
shared.tmsOutPort = factory.createPortInstance()
shared.tmsOutPort.owner = shared.tms
shared.tmsOutPort.name = "out"
StereotypesHelper.addStereotypeByString shared.tmsOutPort, "FlowPort"
SysMLHelper.setDirectionFlowPort shared.tmsOutPort, "out"
```

4.4.4 Constraints and evaluation functions

The evolution graph in figure 8 has eight potential evolution paths. Formalized, automatically checkable constraints and evaluation functions would be helpful for choosing among these paths. Due to the time limitations of my internship, I did not have time to formalize constraints and evaluation functions as part of this case study. However, I did consider informally the sorts of analyses that would be helpful here and how they could be captured in principle.

Many of the concerns pertaining to the alternative evolution paths appear to be based on risk. For example, as I mentioned earlier, the primary argument against evolving directly from the initial system to the target system is that it entails substantial risks. Similar trade-offs are involved in many of the other evolution paths. In section 8.3 on future work, I will argue that risk (or uncertainty) is a special kind of quality that merits special consideration in the context of software architecture evolution. In this case, analyzing risk would likely entail the construction of some sort of probability model for the evolution, so that we could model the likelihood that various contingencies may occur and the effect that those contingencies would have. There is a great deal of existing work on risk modeling, both for the software industry specifically and also in more general contexts such as economics, which we could draw on to develop a model of risk in software architecture evolution. For now, this remains as future work.

Other prominent concerns about this evolution include time, cost, and collaboration. These are somewhat more straightforward to model. Recall that the transitions in our model are composed out of smaller, simpler operators; if we can understand the time and cost properties of these operators, we can compose them to develop time and cost models for the entire evolution graph. Estimating the time and cost of these operators is still not trivial, but it is considerably easier than attempting to understand the entire evolution graph at once.

These are all "business" issues rather than technical ones. But there are also technical constraints in play in this evolution, and technical constraints are often simpler to analyze in purely structural terms than business constraints. In this evolution, we might have a constraint that there are always complete pathways by which commands may be uplinked to the spacecraft and telemetry downlinked; if not, there is a bug in the model.

Thus, even though detailed constraint and analysis definition was not a focus of this case study, in general terms it appears that our approach is well suited to capturing the sorts of constraints and analyses that are relevant to this case.

4.5 Summary

The introduction to this chapter mentioned that this case study had three main goals. I revisit these goals here and discuss what conclusion may be drawn from this work.

1. **Understand a real-world software architecture evolution problem in its natural context.** In this case study, I spent ten weeks at JPL and examined an ongoing software architecture evolution in its real-world context. Section 4.2 described this evolution in detail. This kind of rich description of real-world architecture evolution is valuable for its own sake, because it can help us to understand challenges and practices of architecture evolution as it is carried out today. Unfortunately, there is surprisingly little existing work examining real-world architecture evolution in depth—much existing research on architecture evolution relies heavily on artificial and toy examples (see section 7.4)—so examining actual software architecture evolution as it occurs in situ has great value in informing architecture evolution research and grounding it in reality. But this work is also helpful for evaluating the assumptions underlying our approach to software architecture evolution. The MDAS evolution examined in this case study turned out to be readily comprehensible from the standpoint of our theory of architecture evolution; the evolution could be easily and comprehensively understood in terms of initial and target states, candidate evolution paths, evolutionary transitions, and so on. This is encouraging because it provides some evidence that our model of software architecture evolution, and the assumptions that we made in developing it, are compatible with software architecture as it is practiced in reality.

2. **Assess the usefulness of our framework for software architecture evolution in helping to plan evolutions and reason about trade-offs.** Currently, architects at JPL use no special tools or approaches to help them reason about and plan evolution. Architecture modeling tools are used heavily at JPL, but their function is to represent existing systems, not to plan evolution. Planning for

evolution is accomplished chiefly via requirements documents and informal architectural sketches of target states; there is no tool support for architectural planning.

Through the modeling effort described in sections 4.3 and 4.4, I demonstrated that our approach can capture many of the real-world architectural concerns relevant to this evolution instance. Of course, this applicability result should not be overstated. Aside from the inherit generalization limitations of a single case study, which I will discuss at length in the Costco case study in section 5.6, this case study evaluated only a subset of our modeling approach; constraints and evaluation functions were not formalized. Bearing these limitations in mind, however, this case study does provide evidence that our approach can be applied to the concerns that arise in a real-world software architecture evolution.

3. **Assess the ease of implementing our approach to software architecture evolution with off-the-shelf languages and tools.** Much of our work in this area, including the case study in chapter 4, relies on research languages and tools, particularly Acme and AcmeStudio. This case study shows that benefits in evolution planning can be achieved even without special-purpose, custom tools. Our approach can be adapted to languages and tools already in place at organizations like JPL—such as SysML and MagicDraw—in a straightforward manner. This bodes well for the adoptability of our approach.

This case study is significant because it was the first detailed case study of a real-world software organization in the context of software architecture evolution modeling (see section 7.4 to compare related work). Its results provide support for the applicability claim articulated in section 1.3.3: that our approach can capture the concerns that arise in a real-world evolution.

But although this case study made a significant contribution, it was by no means a final answer to the question of the real-world applicability of an architecture evolution modeling approach. Indeed, a single-case study, by its nature, is an in-depth study of just one organization. While it can provide evidence that an approach is applicable to at least *some* systems, the kind of generalization that is possible in a case study doesn't permit us to draw universal conclusions on the basis of a study of a single organization. (We will examine the topic of case study generalization more formally in section 5.6.3.) Case study results are strengthened greatly when multiple case studies of different organizations produce similar results.

Moreover, even beyond the inherent limitations of a case study, the JPL work had particular limitations that motivated us to pursue a second case study. First, the JPL case study focused specifically on the construction of an evolution graph; due to time limitations, I did not formalize constraints and evaluation functions as part of this case study (see section 4.3.4). Thus, this case study did not help us to evaluate the applicability of evolution path constraints and evaluation functions. Second, the JPL case study focused heavily on issues in adapting our approach to commercial modeling languages and tools (an issue to which I will return in section 6.2). This is one dimension of applicability, but there are other dimensions of applicability worth considering in greater depth than this case study permitted—such as the degree to which the basic concepts in our approach are adequate for capturing the concerns that arise in a real-world evolution. Third, this case study was based on information that was gathered informally during the course of an internship. This sort of ad hoc data collection can be highly useful, particularly in cases such as this one where more elaborate data collection techniques are impractical. However, a more formal and regimented case study design permits generalizations that may not be warranted in a less formal case study such as this one. We thus decided to undertake a second case study with a rather different design, focusing on a different software organization, with more careful and methodical consideration of issues such as validity, reliability, and replicability. I will discuss the differences between the two case studies further in section 5.7.

5 Case study: Architecture evolution at Costco

In this chapter, I discuss the second of the two case studies that together constitute the empirical evaluation of my thesis.[7] I will discuss the differences between the two case studies at length in section 5.7.

This second case study was carried out at Costco Wholesale Corporation, a major U.S.-based retailer, and sought to answer three research questions:

1. How do practicing architects in a real-world software organization plan and reason about evolution?
2. What difficulties do practicing architects face in planning and carrying out evolution?
3. How well can our modeling framework capture the concerns that arise in a real-world architecture evolution?

(These case study research questions are not to be confused with the research questions of my thesis work generally, presented in section 1.1. I will discuss how this case study relates to the questions and claims of my thesis in section 5.7.)

Figure 9 provides an overview of the design of the case study. The first step was developing a case study design. I produced a formal case study design document, following the guidelines provided by Yin [226, ch. 2]. Once the case study was designed, I obtained approval to carry out the case study from Carnegie Mellon University's institutional review board.

After these initial, preparatory stages, the next phase of the case study was data collection. To collect the data for this case study, I spent two weeks at Costco, where I conducted semistructured interviews with architects and examined a variety of architectural documentation. Section 5.2 describes the data collection procedures in detail.

The main analytical method used in the case study was content analysis [195]. In fact, I conducted two content analyses in parallel: one examining research questions

[7]This chapter is an abridgment of a much longer case study report. For additional details on the case study, refer to the full report [17].

I would like to express my sincere gratitude to Shrikant Palkar at Costco Wholesale Corporation for facilitating this case study. I would also like to thank the anonymous interview participants for contributing their time.

The opinions and findings in this document are mine alone and should not be construed as representing the policies or opinions of Costco Wholesale Corporation. Furthermore, where individual research participants are quoted, their comments are their own and should not be interpreted as representing the policies or positions of their employer.

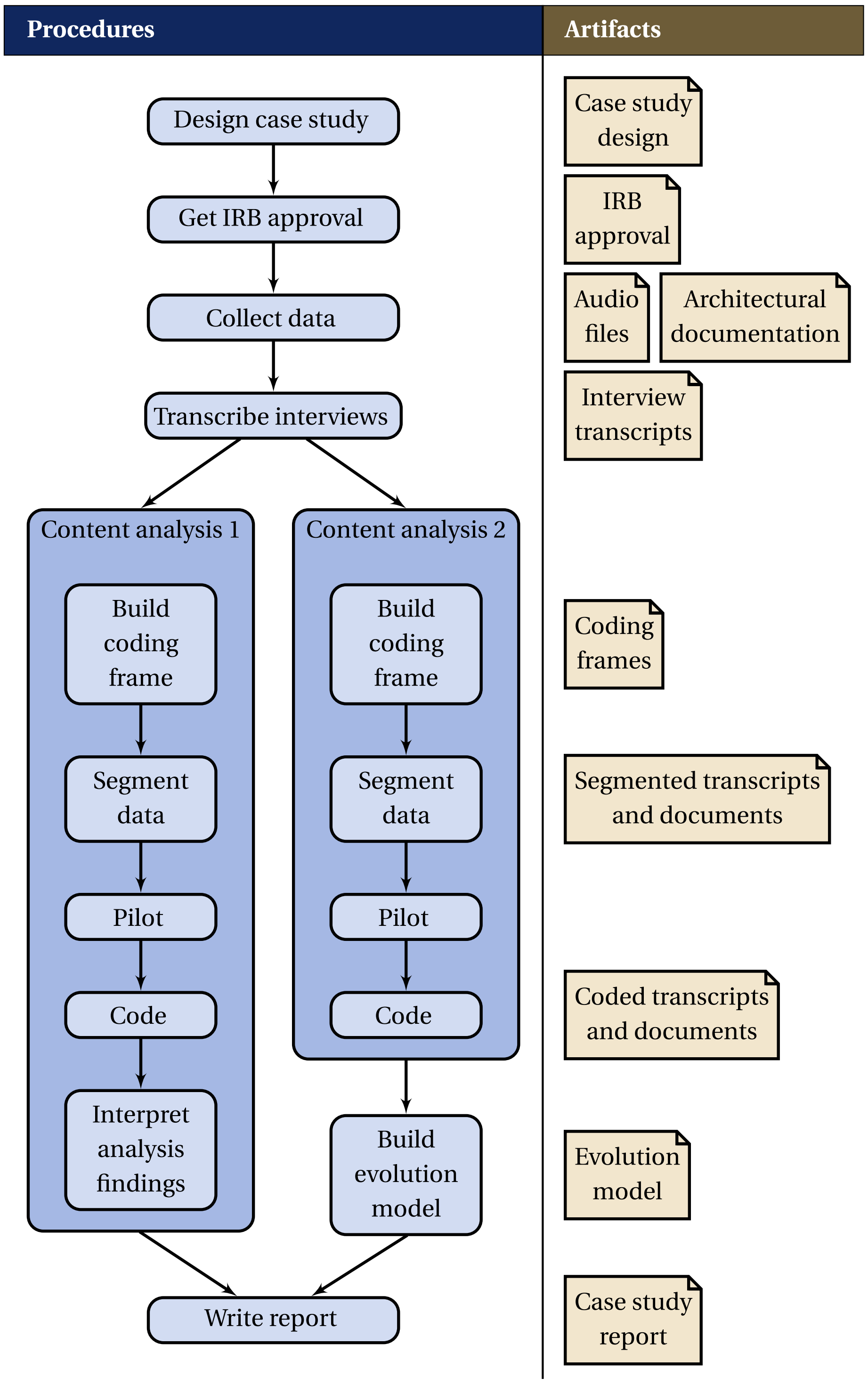


Figure 9. *An overview of the case study design. The stages of the case study procedure appear on the left. Case study artifacts appear to the right of the stage in which they are produced.*

1 and 2 as defined above, and one examining research question 3. I will explain the reasons for this decision in detail in section 5.3.4, but to summarize briefly, the use of two content analyses was motivated chiefly by the different characters of the research questions: research questions 1 and 2 are *descriptive* of the current state of practice of architecture evolution, while research question 3 is *evaluative* of our approach to evolution modeling. (For a discussion of different types of case studies, see Yin [226, ch. 1].) Research questions 1 and 2 could be answered directly through a content analysis of the interview data, while research question 3 was answered through a two-stage analysis: a content analysis followed by a modeling phase. The content analysis yielded key architectural elements and evolution elements in the research data, and these results fed into a modeling phase in which I constructed a (partial) evolution model using our approach. The content analysis is explained in detail in section 5.3, and the modeling phase is described in section 5.4.

Finally, the conclusions of the case study are discussed in sections 5.5, 5.6, and 5.7. Section 5.5 discusses the findings of content analysis 1, and section 5.6 discusses the overall conclusions of the case study (including both content analyses as well as the modeling procedure) with respect to the research questions defined above. Section 5.6 also discusses issues of reliability and validity. Section 5.7 explains the differences between the two case studies described in this dissertation and their relevance to the thesis.

This chapter adheres to what Yin [226, p. 176] calls a linear-analytic case study report structure: the problem is introduced, the methods are described, the findings are reported, and then conclusions are drawn.

I should note that the final case study report (on which this dissertation chapter is based) was reviewed by Costco, which requested that I remove or edit certain passages containing information that they deemed to be confidential or sensitive. For example, the company requested that I remove all references to specific vendors associated with the company. I complied with all such requests. All of these changes were fairly minor and did not, in my judgment, materially influence the presentation of the overall findings of the case study.

5.1 Case selection

In this section, I describe the organization that served as the case for this case study. Section 5.1.1 gives general background on the company and explains why it was a suitable choice for this case study. Section 5.1.2 describes its architectural organization.

5.1.1 About the case

Costco Wholesale Corporation, founded in 1983, is the second-largest retailer in the United States [211] and the sixth-largest in the world [210]. Currently the company has about 627 locations worldwide, of which 449 are in the United States [59]. The company uses a warehouse club business model, meaning that it sells goods in large, wholesale quantities; thus, you can buy a 30-pack of toilet paper, but (unlike

Wal-Mart or Target) not a 4-pack. Retail locations are called "warehouses" rather than "stores" and have a spartan decor, with concrete-floor aisles flanked by stacks of cardboard boxes. The company uses a membership-only model; only people who have purchased a membership may shop there. There are currently about 70 million members [59].

The company's no-frills warehouses, volume purchasing, and membership model are part of what allows it to keep prices lower than competitors while remaining profitable. Other contributing factors are the company's efficient distribution network, low marketing expenses, limited warehouse hours, rapid inventory turnover, and low payment processing fees (warehouses don't accept credit cards other than American Express). Warehouses also have far fewer distinct items than typical discount retailers, greatly simplifying inventory management. A typical warehouse carries around 3,500 SKUs [58]; by comparison, a typical Wal-Mart Supercenter has 142,000 [34].

Over the years, Costco has accumulated a patchwork of legacy software systems. Until recently, the company had always built almost all of its software systems, even systems that are more typically purchased as off-the-shelf packages, such as the company's core financial systems. A few years ago, the company embarked on an extensive and prolonged modernization effort to revamp many of its key software systems, which were growing archaic. A few of these systems were simply rebuilt in modern languages with modern engineering techniques, but for the most part, the company has transitioned from its tradition of homegrown software to a posture of buying off-the-shelf software whenever possible. Thus, many of the company's old, custom-built systems are being replaced with off-the-shelf products. As a result of this far-reaching modernization effort, architecture-level evolution of software systems is now pervasive there. The contemporaneous overhaul of so many core systems has posed significant integration and communication challenges. These factors made the company an appealing candidate for a case study on architecture evolution.

5.1.2 The architecture organization

Understanding the organizational structure of the architecture group that was the focus of this case study will be helpful for contextualizing the rest of this chapter. The architecture group underwent a significant reorganization several years ago, at the beginning of the modernization effort just described. Until recently, architects were scattered throughout the IT department; people in architectural roles were associated with particular teams or projects, and there was no central architecture organization, nor even regular communication among the architects. This made it difficult to understand the dependencies among systems and to diagnose integration problems, so a central enterprise architecture group was established to define and maintain enterprise standards and to manage and harmonize architectural practices across the organization.

In figure 10, I have diagrammed my understanding of the current state of the architecture organization, based on the interviews I conducted. At the top of the architecture organization are four enterprise architects (EAs). Each of these four

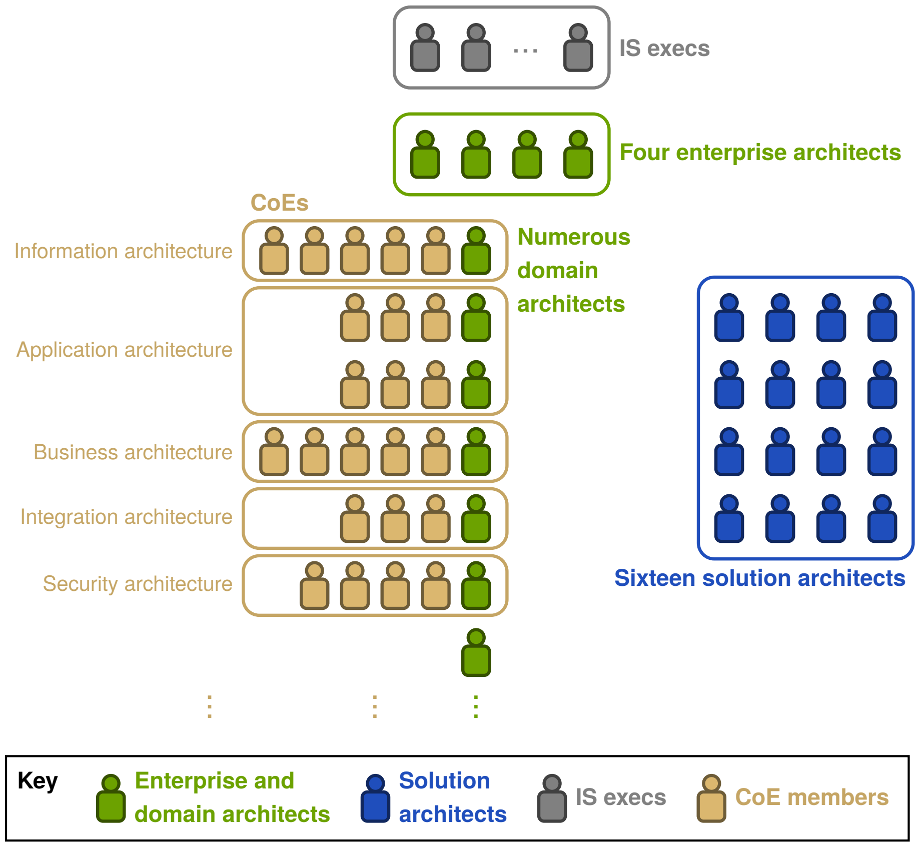


Figure 10. *The organizational structure of Costco's architects.*

individuals has a broad set of responsibilities, but they all have their own focuses. One of them, for example, is primarily responsible for infrastructure; another has oversight over process issues. The EAs are together responsible for defining strategy and making policies that affect the organization as a whole. They are tasked with ensuring that the company's software systems are architecturally consistent—to guard against the kind of architectural incongruities, integration troubles, and communication problems that beleaguered the company years ago, before a central architecture organization was created. One of the ways they do this is by leading the Enterprise Architecture Review Board, which must approve major architectural decisions to ensure that they are in conformance with corporate strategy and practices.

Under the EAs are a number of domain architects. These domain architects are also doing enterprise architecture, in the sense that they also set standards and provide guidance that is applicable to the organization as a whole. Each domain architect has responsibility over a single domain of expertise; thus, there are information architects, infrastructure architects, business architects, application architects, security architects, integration architects, and so on. (For a discussion of different roles and domains within an enterprise architecture organization, see Winter & Fischer [223].)

When this new enterprise architecture group was formed, there was concern that it would become an ivory tower, estranged from the experiences of the people actually building the software, dispensing impractical guidance from on high. To avoid this, centers of excellence (CoEs) were established as a way of keeping the work of the domain architects grounded in reality. Each domain architect, generally speaking, leads a CoE that has some decision-making responsibility within that architect's domain. Thus, the business architect leads a business architecture CoE, an application architect leads an application architecture CoE, and so on. The CoEs are led by the domain architects, but they are staffed by engineers and other personnel who have hands-on roles in building, maintaining, or managing systems. These people have practical, day-to-day domain expertise that complements and grounds the domain architect's broader, theoretical domain expertise. For example, the business architecture CoE, which is led by the business architect, is staffed by a business process analyst, a data steward, a business analyst, a service engineer, and a couple of product analysts.

Together, the four EAs and the domain architects form the enterprise architecture side of the architecture organization, support by these CoEs. The other side of the architecture organization is the solution architects, who are responsible for specific projects. A solution architect is usually involved with a few projects at a time. It is the solution architect who actually creates the architectural designs for individual projects, in accordance with the practices approved by the enterprise architecture group, and with guidance from domain architects with relevant expertise. The solution architect gathers project requirements, defines the high-level design for a project (which will be validated by the enterprise architecture review board), hands it off to a lead developer for lower-level design, and oversees architectural matters as the solution is implemented and released. There are sixteen solution architects in total.

Together these three groups—enterprise architects, domain architects, and solution architects—constitute the architecture group. The EAs provide leadership and define strategies and policies that affect the entire organization, the domain architects work with their CoEs to define practices and provide guidance within their respective domains, and the solution architects work in support of individual projects.

5.2 Data collection

Data collection was carried out during a two-week visit to Costco's headquarters in Issaquah, Washington. I gathered two types of data: interview data and written architectural documentation.

5.2.1 Interview data

The most important source of data was a series of research interviews that I conducted. During my visit to Costco, I interviewed six participants in eight interview sessions. Some participants were interviewed more than once, and also in some interview sessions multiple participants were present.

The case study adhered to a *semistructured* interviewing discipline, which means that although the interviews were guided by an explicit interview protocol that defined the general topics that the interviews would examine, I was free to devise new questions on the fly to explore interviewees' knowledge at will. This is in contrast to a *structured* interview, in which every question is entirely scripted and the interviewer is strongly constrained, or an *unstructured* interview, in which the interviewer is completely unconstrained and the interview has no set format.

The choice of semistructured interviewing was the most appropriate for this type of research, as I had some familiarity with the domain and a strong sense of the kind of information I was seeking, but I needed the freedom to explore participants' responses and investigate closely those topics on which a particular participant might have especially deep knowledge. Unstructured interviews are most useful for exploring a topic about which very little is known, while structured interviews are most useful when the focus of the research is on directly comparing responses across participants (as is generally the case in quantitative interview research). For further guidance on conducting research interviews, see Kvale & Brinkmann [125].

The interview protocol is reproduced in appendix A. It includes an introductory script to secure informed consent followed by a series of topics to be covered: the participant's background, the practice of architecture evolution, limitations of current approaches to managing evolution, and specifics about the evolution of particular software systems. Interview durations ranged from 17 minutes to 56 minutes.

I captured audio recordings of all interviews. Recording interviews has a number of advantages. First, it provides the researcher with an accurate and complete record of each interview, so that the researcher is not forced to rely solely on his own notes and memories, which are inevitably inaccurate and incomplete. Second, it frees the interviewer from taking excessive, meticulously complete notes during the course of the interview, which can encumber the flow of dialogue. Third, it bolsters the validity of the research; when a researcher has only his memories and notes to go on, it is easier for him to impose his own biases on interview responses. Fourth, it enables transcription and content analysis of interview responses.

I fully transcribed all eight of the interviews I conducted. Transcription allows much easier reference to the contents of an interview, permitting the researcher to peruse the collected interview data freely without the need to listen to long stretches of audio to find the desired information. Indeed, unlike an audio recording, a transcript is fully searchable, allowing the researcher to instantly find significant passages. Transcription also enables various kinds of analysis, including content analysis (section 5.3).

In the social sciences literature, there is a great deal of discussion on methods and best practices for transcription of research interviews. One of the most basic decisions that a researcher must make when transcribing interviews is whether to try to capture all the nuances and quirks of verbal communication, such as pauses, pronunciation idiosyncrasies, stutters, false starts, and nonverbal utterances, or whether to instead adapt the material to the written form, eliding stutters and verbal tics, standardizing grammar, and adding punctuation. In the literature, these are sometimes called *denaturalized* and *naturalized* approaches to transcription[40].[8] Of course, these are really two ends of a spectrum, and often it is most appropriate to take a middle course. But in the extreme case, highly naturalized transcriptions can read much like written, edited prose, while highly denaturalized transcriptions often adopt abstruse notation systems to capture the nuances of spoken language, resulting in a transcript that bears little resemblance to ordinary natural language.

This choice is not an inconsequential one. A denaturalized transcription system includes more information, but it does so at the risk of obscuring the content. A naturalized transcription system reads much more easily, but it does so at the cost of losing information that might inform interpretation of the text. These issues have long been a topic of debate, but a good general principle is that such choices should be driven by the purpose of the analysis [143; 161; 168; 185; 193]. In this case, we are interested in the interviews solely for their information content; we are interested in what is said, not how it is said. As a result, I adopted a highly naturalized form of transcription.

The transcription process itself was methodical and painstaking. I listened to each interview several times to ensure transcription accuracy and minimize the number of passages that had to be marked inaudible. (I used two different recorders to capture each interview. Because the two recorders were differently positioned, each picked up some utterances that the other did not, so I used both recordings in the transcription process.) After all eight interviews were transcribed, I went through a final editing phase to ensure consistent orthography across interviews. In total, transcribing all eight interviews took approximately fifty hours of work.

5.2.2 Architectural documentation

In addition to interviewing personnel, I was also permitted to access many architectural documents pertaining to the company's software systems. These were a very useful source of data and particularly useful as a complement to the interview data that I collected.

Particularly significant examples of documents to which I had access include:

- Thirty-five "architectural decision" documents, each of which captured an architectural decision that had been approved by the enterprise architecture team in accordance with a defined process. Each of these documents described the background and justification for the decision, as well as alternatives considered.

[8] Unfortunately these useful terms have been compromised by inconsistent usage. When Bucholtz [40] introduced the terms in 2000, she defined naturalized transcription as transcription that privileges written over oral discourse features, producing a "literacized" text. Denaturalized transcription, in Bucholtz's original terminology, was transcription that prioritizes faithfulness to oral language through the use of verbatim transcription, including discourse markers, repetitions, repairs, and so on. However, in a 2005 paper, Oliver et al. [168] reversed these terms (perhaps unintentionally), using the term *naturalized* to describe what Bucholtz had called denaturalized transcription and vice versa. Usage of the terms in the literature is now inconsistent, with some authors [111; 161] using Bucholtz's original meanings for the terms and others [152; 157] following Oliver et al. Here I do the former.

- Twenty-one "strategy" documents describing, in more general terms, strategic initiatives from an architectural standpoint.
- Two detailed architectural design documents describing a specific evolution, the evolution of Costco's point-of-sale system, which would become the focus of the modeling phase of the case study (section 5.4).

These documents were useful because they provided clear, explicit articulation of the architectural reasoning and decision-making process. However, they did not provide detailed historical information. Thus, these documents were most useful as a complement to the interview data, which provides a better historical perspective.

5.3 Content analysis

The main analytical method used in this case study was *content analysis* [195]. Content analysis is a research method, commonly used in the social sciences, for extracting meaning from text. A fairly standard definition of *content analysis* is that given by Krippendorff [122, p. 24]: "a research technique for making replicable and valid inferences from texts (or other meaningful matter) to the contexts of their use."

Content analysis has been seldom used in software engineering research (e.g., [3; 9; 165]) and almost never used in software architecture research. Because of this, many readers may be unfamiliar with it, so I present here an overview of the method and a fairly detailed discussion of relevant methodological considerations in adopting it.

Section 5.3.1 presents a brief history of content analysis. Section 5.3.2 discusses the difference between qualitative and quantitative content analysis and explains why this case study uses the former. Section 5.3.3 describes the elements of a qualitative content analysis. Finally, sections 5.3.4–5.3.7 describe how content analysis was used in this case study.

5.3.1 A very brief history of content analysis

The history of content analysis can be traced back centuries; for an overview of this early history, see Krippendorff [122, ch. 1]. Content analysis in its modern form, however, came into being during the Second World War, when the renowned social scientist Harold Lasswell, acting as the chief of the United States government's Experimental Division for the Study of War-Time Communications, applied content analysis to Axis propaganda [190, ch. 6]. Lasswell standardized techniques that are still used in content analysis today, such as pilot testing, the use of formal sampling criteria, and assessments of reliability based on interrater agreement. In 1952, Bernard Berelson published the first major textbook on content analysis [27].

The ensuing decades saw ever-increasing adoption of content analysis by researchers [164, ch. 2]. The method was adapted to varied applications in many different disciplines, from readability analyses [85] to authorship attribution [160] to studies of television violence [57]. Today, content analysis is a major research method in a number of disciplines, and many textbooks and articles give guidance on the method and its application.

5.3.2 Qualitative versus quantitative content analysis

Content analysis has spawned many variants as it has been applied to many different fields, but the oldest and most enduring division is between *quantitative* and *qualitative* content analysis. Defining the terms *quantitative* and *qualitative* precisely is difficult [73]. There is no canonical, accepted definition of qualitative research. But a good definition that conveys many of the main points is that of Cassell & Symon [44, p. 7], who distinguish between qualitative and quantitative methods by defining qualitative research as being characterized by:

> a focus on interpretation rather than quantification; an emphasis on subjectivity rather than objectivity; flexibility in the process of conducting research; an orientation towards process rather than outcome; a concern with context—regarding behaviour and situation as inextricably linked in forming experience; and finally, an explicit recognition of the impact of the research process on the research situation.

Schreier [195, pp. 15–17] identifies seven key differences that distinguish qualitative content analysis from quantitative content analysis: a focus on latent rather than manifest meaning, a greater need to take context into account during the analysis, variable handling of reliability, an increased focus on validity, the use of coding frames that are at least partly data-driven, a tendency to be used to make broader inferences, and greater variability in its process. We will return to some of these points shortly.

Content analysis was originally introduced as a quantitative method. In his seminal 1952 textbook, Berelson defined content analysis as "a research technique for the objective, systematic, and quantitative description of the manifest content of communication" [27, p. 18]. But it did not take long at all for critics to find fault with this definition, particularly with the words *quantitative* and *manifest*; in the very same year, Kracauer published a critical response to Berelson, entitled "The Challenge of Qualitative Content Analysis" [119], that argued that meaning is often complex, context-dependent, and manifest, and that quantitative approaches are insufficient for analysis of such meaning.

This interchange defined the contours of a debate that continues to this day. Quantitative content analysis has retained its status as the dominant form of the method, and there has been an ongoing debate about the rigor, validity, and appropriateness of qualitative alternatives. Advocates of qualitative content analysis argue that a qualitative approach is useful and necessary for analyzing the content of texts where meaning is highly subjective and context-dependent and quantitative measurements are ill suited to describing meaning. Meanwhile, critics of qualitative content analysis have expressed skepticism about its reliability and its methodological rigor.

Indeed, to a significant degree, mainstream quantitative content analysts continue to be fairly dismissive of qualitative content analysis. Krippendorff's *Content Analysis: An Introduction to Its Methodology*, the most popular text on the method, equates qualitative content analyses with what Krippendorff calls "text-driven content analyses" [122, § 14.1.1]—analyses that are motivated by the availability of texts rather than by epistemic research questions—which he dismisses as “fishing expeditions” [p. 355]. “Qualitative content analysts,” he writes, “typically stop at their own interpretations of texts” [p. 357]. However, he grants that “qualitative approaches to text interpretation are not incompatible with content analysis” [p. 89].

Neuendorf’s *The Content Analysis Guidebook*, another major text on the method, is even more dismissive, devoting only one sentence to the topic of qualitative content analysis: “Although some authors maintain that a nonquantitative (i.e., ‘qualitative’) content analysis is feasible, that is not the view presented in this book” [164, p. 14].

It may be true that early examples of qualitative content analysis lacked some of the methodological rigors of the quantitative method. However, great strides have been made in developing qualitative content analysis into a rigorous research methodology with a disciplined process and with careful consideration of validity and reliability.

The methodologist most responsible for these strides is Philipp Mayring, who in 1983 published (in German) an influential textbook on qualitative content analysis that standardized the discipline and introduced methods for evaluating reliability and validity [150]. However, it was not until 2000 that any of Mayring’s writing on qualitative content analysis was translated into English, and even then only a short journal article [148]. (The textbook remains untranslated to date.) As a result, qualitative content analysis was, for a long time, much more widely adopted by German researchers than by Anglophones.

This is now changing. Adoption of qualitative content analysis in the English-speaking research community has increased in recent years and is likely to continue increasing. One of the first English-language books on qualitative content analysis was published last year by a German methodologist, Margrit Schreier [195].

Qualitative content analysis, in its modern, systematized form, has a number of qualities that make it more suitable than quantitative content analysis for this case study:

- Qualitative content analysis is intended for use in cases where *interpretation* is necessary to analyze the data—when the meaning of the material is not standardized, is context-dependent, or would likely be understood differently by different readers. A different way of putting this is that qualitative content analysis specifically seeks to examine the *latent* meaning of texts rather than only the *manifest* content. (Quantitative content analysis is often applied to latent content as well, but qualitative content analysis is designed specifically to facilitate reliable and explicit interpretation of texts.)

  Schreier [195, p. 2] explains this point with an example based on analysis of magazine advertisements. If a study seeks to find out information about the number of women and men who appear in magazine advertisements, then quantitative content analysis would be more suitable than qualitative content analysis, because very little interpretation is required to determine whether the persons in a picture are female or male. But if a study seeks to determine whether women in magazine advertisements are more likely to be placed in trivial contexts than men, qualitative content analysis would be a highly suitable method, because there is a significant interpretative step required to determine whether a given context is "trivial" (different readers might well disagree). Qualitative content analysis provides a method for managing this ambiguity in a valid and reliable way.

In the present study, we are explicitly concerned with interpretation of the language that architects use in their discourse on evolution. The meaning of architectural terms is not sufficiently standardized to justify a classical approach in which we simply apply our own unexamined interpretations to the data; indeed, even basic terms like *architecture* and *evolution* mean very different things to different people. The use of qualitative content analysis allows us to deal in an explicit and rigorous way with such polysemy and helps us to avoid imposing our own biases onto the interpretation of the data.

In addition, one of the particular goals of this study is to translate, in a replicable way, the language that real-world architects use when talking about evolution into the modeling constructs of our approach to architecture evolution. This is merely a special kind of interpretation, and qualitative content analysis can help us with it by providing a systematic method for resolving ambiguities to extract meaning from text.

- In quantitative content analysis, the research questions are addressed via frequency counts (or quantitative measures derived from frequency counts through methods such as cross-tabulation or factor analysis); counting things (words, codes, categories, etc.) is almost invariably a core part of the analysis.

  In qualitative content analysis, frequency counts and statistical methods *may* be used (and often are)—after all, the method necessarily involves unitizing the material and coding it based on a finite set of predefined categories, so it's quite natural to tally the results once the material is coded. But in qualitative content analysis, it is equally acceptable to conduct a textual (nonfrequency) analysis, using the codes and categories as a way to achieve reliability and avoid bias, rather than as figures to be tallied.

  The research questions that interest us in this study are not quantitative in character (although frequency data may be helpful in answering them—and indeed we will make use of frequency data in our analysis). In addition, a quantitative analysis approach is most appropriate when there are many subjects to be compared—when the goal is to compare or relate results collected from many individuals (individual people, individual companies, individual texts, etc.). Our content analysis, on the other hand, is conducted within the context of a case study—indeed, a single-case study. There is only one "subject" of the case study: Costco. (There were six human participants—the interviewees—but they are not the subjects of the case study. Rather, their role is as informants who can supply data about the true subject of the study, Costco.)

- Qualitative content analysis has a more flexible method than quantitative content analysis. In quantitative content analysis, the procedure is fairly fixed. There are many variants of the method, but each variant has a prescribed set of steps, with only certain kinds of deviations permitted.

Qualitative content analysis has historically been much looser. Critics of qualitative content analysis have regarded this as a flaw, while advocates have regarded it as a strength. Here, we are using a particularly well-defined form of qualitative content analysis that does have a process with a fixed set of steps. But even so, much more variation is permitted in the execution of those steps than in classical quantitative analysis. For example, the coding frame may be defined in a very data-driven way, or in a very concept-driven way; reliability may be assessed by a variety of different methods [195, p. 17]. Such flexibility is very useful in the present case study, as we will see later in this chapter.

In reality, the qualitative/quantitative division in content analysis is not a dichotomy, but a continuum [105]. Many qualitative content analyses have very quantitative aspects, and many classically quantitative content analyses deal with latent meaning, involve rich textual interpretation, and adopt methodological variations. However, the above discussion makes clear that qualitative content analysis is much more suitable for our research questions than quantitative content analysis.

5.3.3 Elements of a qualitative content analysis

Schreier [195] describes a qualitative content analysis as comprising these main steps:

1. **Define the research questions.** The research questions for this case study are given at the beginning of this chapter.

2. **Build a coding frame.** Coding—an analysis technique in which researchers annotate segments of text with descriptive codes, categories, or labels—is probably the most well known and widely used of all qualitative analysis methods. There are many methods for coding, which vary greatly in their purpose, use, and approach [192].

   In qualitative content analysis, coding is used as a way of reducing the data and focusing the analysis on the research questions. Coding is done in accordance with a coding frame, which defines the codes that may be applied to segments of data. The construction of the coding frame circumscribes the analysis. Content analysis is a reductive process; the coding frame determines which information will be excluded from the analysis and which is important enough to be included.

3. **Divide the material into coding units.** In content analysis, codes are not applied at will to arbitrary segments of text, as in some coding methods. Rather, the text is first divided into *coding units*, such that each unit may be coded with at most one subcategory of a main category in the coding frame.

   This is done for a few reasons. First, it forces the researcher to analyze all the material, segment by segment, avoiding the risk that the researcher will inadvertently ignore portions of the text or devote too much attention to the passages that are most striking (or that fit into a preconceived narrative). Second, it helps to keep the analysis directed toward answering the research questions. Third, it facilitates double-coding as a means of demonstrating reliability.

4. **Try out the coding frame.** Before the main coding phase begins, a pilot phase is recommended in which the researcher applies the coding frame to a portion of the material. This can reveal problems that would otherwise crop up during the main analysis.
5. **Evaluate the trial coding and finalize the coding frame.** After the pilot phase, the coding frame is evaluated and modified as necessary in preparation for the main analysis.
6. **Carry out the main analysis.** Coding in the main analysis is conducted much as in the pilot phase, except all material is now coded.
7. **Interpret the findings.** There are various techniques that can be applied to further analyze the results of a content analysis, and various ways that the findings of a content analysis can be presented. The ultimate goal is to answer the research questions in a way that is valid and reliable.

The following sections describe how these steps were carried out in this case study.

5.3.4 Coding frame

The coding frame that I developed is reproduced in its entirety in appendix B. In this section, I explain how it was constructed.

The construction of the coding frame is one of the most crucial steps in a content analysis. After all, the coding frame defines the rules that govern the interpretation of the text. Thus, the construction of the coding frame defines the shape of the rest of the content analysis, including the segmentation of the text into units, the coding itself, and the tabulation and reporting of the findings.

There is a particular structure to which a coding frame for a qualitative content analysis should adhere [195]. The starting point for a coding frame is a set of *main categories*, which define the dimensions or aspects on which the analysis will focus. Within each main category is a set of *subcategories* that specify what may be said about the aspect represented by the main category. In effect, the main categories are analogous to the variables in a quantitative study, and the subcategories are analogous to the levels of those variables. Coding frames vary greatly in complexity; there may be one or many main categories, and there may also be multiple levels of hierarchy, with subcategories containing their own further subcategories. (Note that in content analysis, *category* is effectively synonymous with *code*, even though these terms are importantly different in, e.g., grounded theory.)

For the present work, it is useful to observe that the research questions stated at the beginning of this chapter are of two very different characters. Research questions 1 and 2 are questions about how evolution happens in the real world today; these questions are *descriptive* of architecture as it is practiced today. They will be answered directly through a content analysis of the interview data.

Research question 3 is quite different; it asks whether our modeling approach is suitable for representing the concerns of a real-world evolution. This research

Content analysis	Content analysis 1	Content analysis 2
Research questions	1 and 2	3
Purpose	Descriptive	Evaluative
Material	All interview data	Interview data and architectural documentation pertaining to the point-of-sale evolution
Coding frame	Data-driven	Concept-driven
Segmentation	A coding unit is a passage of text addressing one topic of interest	A coding unit is a word or phrase representing a particular architectural element
Use	Results will be directly interpreted with respect to the research questions	Results will feed into a modeling phase to evaluate applicability

Table 2. *A summary of the differences between the two content analyses used in this case study.*

question is *evaluative* of our approach. It will be answered via a two-step process in which we first apply content analysis to that portion of the research data which pertains to a specific evolution (the evolution of the point-of-sale system), then use the results of that content analysis to construct a model of the evolution in accordance with our approach.

It is therefore useful to describe the analysis as comprising two entirely separate qualitative content analyses, one (which I will call “content analysis 1”) targeted at the descriptive research questions and one (“content analysis 2”) targeted at the evaluative research question. A summary of the two content analyses appears in table 2.

The differences between these two content analyses necessitate that their coding frames likewise be constructed differently. It is helpful here to consider a distinction that is drawn in the methodological literature on qualitative content analysis. Both Mayring [148; 149] and Schreier [195, pp. 84–94], in their treatments of qualitative content analysis, identify two main approaches for developing a coding frame: *data-driven* (or inductive) and *concept-driven* (or deductive) category development. With a data-driven strategy, categories are based on the collected data—developed through progressive summarization of relevant passages of text or other similarly bottom-up strategies. With a concept-driven strategy, categories are defined a priori, without reference to the data. Instead of being derived from the text, categories are based on preexisting theory, prior research, the format of the interviews, or other similar considerations. Most qualitative content analyses, Schreier suggests, will use a combination of data-driven and concept-driven strategies. Schreier emphasizes that the strategy adopted for developing a coding frame should be one that is suited to the goals of the analysis. However, as a general recommendation, she writes that a data-driven strategy is most appropriate for detailed description of material, while a concept-driven strategy is most appropriate for testing hypotheses or establishing comparisons with prior work [195, pp. 105–106]. The distinction between data-driven and concept-driven content analysis might be compared with the difference between the Glaserian and Straussian paradigms in grounded theory; for a discussion of these two grounded theory methods and their application to software engineering, see van Niekerk & Roode [219].

I adopted a principally data-driven strategy for content analysis 1 and a principally concept-driven strategy for content analysis 2. For the descriptive research questions, the purpose of the content analysis is to describe architects' perceptions and experiences regarding architecture evolution. Because the purpose of this content analysis is to describe the material in detail, a primarily data-driven approach is most suitable.

Therefore, after defining the top-level categories based on the research questions ("Evolution motives," "Challenges," etc.), I defined their subcategories using a data-driven strategy. To do so, I made a pass through the entirety of the interview data, marking and paraphrasing any passages that appeared relevant to any of the top-level categories. For example, the following is an excerpt from an architect's reflection on a previous effort to implement approaches that he and his colleagues had read about in well-known books on service-oriented architecture:

> One of the problems that I've seen is that it is very cumbersome. Very good, very sophisticated, but when you are new to this, when you are really not intelligent enough to make those decisions, make those choices, when you have to go through a very elaborate process, it can be very counterproductive. And indeed that's what we saw. When this method is introduced to the world, there's a lot of confusion about: Why do we have to do all these things? I personally participated in some meetings too and saw this whole process as very cumbersome, very confusing, and frankly the result of that is not very good.

Next to this passage, I wrote, "challenge: available approaches cumbersome." After marking all the interview data in this way, I consolidated topically similar marked passages into subcategories. For example, the above passage was combined with several others, marked with paraphrases such as "challenge: no resources for information" and "challenge: lack of needed tools," into a category called "Challenges: Inadequate guidance and tool support."

For content analysis 2, on the other hand, concept-driven category development is appropriate. The goal of this content analysis is to evaluate the suitability of a particular, preexisting model with respect to a specific evolution. This preexisting model forms the basis from which the coding frame is derived.

Since its output will be used to produce an evolution model, content analysis 2 must serve to identify the elements that will appear in the evolution model. Thus, the coding units in content analysis 2 are descriptions of the *architectural elements* (components, connectors, etc.) and *evolution elements* (constraints, operators, etc.)

that will appear in the model. (The concepts in the coding frame and their definitions appear in appendix B.2.2. The application of the coding frame will be discussed in sections 5.3.5–5.3.7; in this section I discuss the definition of the coding frame itself.) The first output that the content analysis must yield is the proper identification of these elements. That is, for each element described by a coding unit, we want to determine through the content analysis how that element should appear in the model—whether it should be characterized as a component, connector, constraint, operator, or some other type of element. Thus, the first part of our coding frame is a classification scheme for elements of the architecture evolution, with categories such as "Classification: Component," "Classification: Constraint," and so on.

We also want to determine whether each identified element should appear in the initial and target state of the evolution model. Thus, the second output that content analysis 2 must yield is a determination of which phases of evolution they appear in: the initial architecture, the target architecture, neither, or both. The second part of the coding frame for content analysis 2, therefore, comprises the categories "Presence in initial architecture: Present," "Presence in initial architecture: Absent," "Presence in target architecture: Present," and "Presence in target architecture: Absent."

The advantage of using content analysis to guide the construction of the model (rather than just constructing a model based on informal impressions and methods, as is more typical in software architecture research) is that the model will be directly and reliably tied to the research data, giving us more confidence that the conclusions we draw from the model are adequately supported by the data and are not influenced by researcher bias. Because the segmentation (i.e., the identification of model elements) and coding (i.e., the classification of model elements) are conducted according to well-defined procedures, we can be more certain that, for example, a model element that we identify as a connector really does represent an actual connector in the system as described in the research data, and that we haven't inadvertently omitted or overlooked elements that ought to appear in the model.

The coding guide itself was written in a format typical of coding guides for content analysis. (For a discussion of the components of a good coding guide, see Schreier [195, pp. 94–104].) For each category, the coding guide defines: the name of the category, an abbreviation for the category (to be used when marking the coding sheet), a detailed description of the category (including an explanation of what it encompasses and guidelines for determining inclusion in the category), and usually an example of the category and sometimes a list of indicators to serve as decision aids. The full coding guide appears in appendix B, as noted earlier.

5.3.5 Segmentation

With the coding frame defined, the next step is to segment the material into *coding units* to which the categories of the coding frame can be applied. As with the construction of the coding frame, the segmentation of the material is handled separately for content analysis 1 and content analysis 2. For content analysis 1, the interview material was segmented *thematically*. That is, the material was divided into segments of sufficiently fine granularity that each coding unit pertained to one topic—where

Researcher: When you say you generally have an idea of where you want to go and how you want to get there—you just can't necessarily execute on it as easily as you'd like to—how do you develop that plan? How do you figure out where you want to go and how you want to get there? Do you have processes to help you, or is it mostly just intuition and experience?

Participant: ^{A9}⟨Mostly intuition and experience, yeah.⟩ ^{A10}⟨You look to the industry to see what's going on⟩, ^{A11}⟨but ultimately, in this particular line of business—and I've worked in software development companies and retailers and laboratory environments, and I can tell you that in this business, you tend to lean toward simplicity. The next system you build is going to last twenty years, and it needs to be maintainable, and everybody needs to be able to capitalize on it and expand and extend when necessary, so you really don't try to get too crazy with it. We're not launching shuttles here, we're selling mayonnaise and toilet paper, so let's keep it in perspective.⟩

A9: Approaches: Experience and intuition

A10: Approaches: Industry practices

A11: Approaches: Rules of thumb and informal strategies

Figure 11. *A segmented and coded passage from an interview transcript.*

topic is interpreted with respect to the coding frame, so that one category is applicable to each coding unit. Figure 11 shows an interview segment as it was segmented and later coded. The angle brackets in the figure identify coding units; observe that the text is split such that each coding unit pertains to a single category of the coding frame.

Segmentation was much more complex for content analysis 2. The objective of content analysis 2 was to identify the architectural elements and evolution elements described in the source material—that is, to identify and distinguish among components, connectors, evolution operators, evolution constraints, and so on, so that they could be included in an evolution model based on the content analysis.

One challenge that this posed was how to deal with multiple mentions of the same element. To correctly identify and code an element, we must consider *all* its mentions throughout the source material. This stands in contrast to the piecemeal interpretative process that is more typical of content analysis, in which we proceed through the relevant passages of a text in sequence, coding each passage in isolation.

Nonetheless, the need to consider multiple mentions of a referent together in order to determine a code is not incompatible with content analysis; it merely requires more careful selection of the coding units. In particular, we can consider multiple mentions of a single referent (e.g., all references to the corporate integration architecture) to together constitute a single coding unit—one that happens not to be contiguous. Krippendorff [122, § 5.2.2] discusses situations where noncontiguous coding units may be helpful:

> The text included in any one recording unit need not be contiguous. Suppose an analyst samples fictional narratives with the aim of studying

> the populations of characters occurring in them. [...] In a typical narrative [...], the characters are rarely dealt with one at a time, or one per paragraph, for example. They tend to interact and evolve over the course of the narrative, and information about them emerges in bits and pieces, often becoming clear only toward the end. To be fair to the nature of narratives, the analyst cannot possibly identify one unit of text with each character. Thus information about a recording unit may be distributed throughout a text. Once the analyst has described the recording units, it is these descriptions, the categories to which they are assigned, that are later compared, analyzed, summarized, and used as the basis for intended inferences.

(*Recording unit* is another term for *coding unit*.) An architectural element or evolution element is similar, in this respect, to a character in a narrative. It may be mentioned many times throughout the material, in a variety of contexts that provide different information about it, all of which must be understood together to obtain a complete view.

Another segmentation challenge posed by content analysis 2 is that not all of the "text" is strictly textual; in addition to the interview material and the written documentation, there are a number of diagrams that must be considered in the analysis. Fortunately, this is not a real problem as far as content analysis is concerned. Content analysis has long been used for analyzing more than just prose. It has been applied to everything from films [60] to photos of models in magazines [74] to children's drawings and cereal boxes [153]. (For general guidance on content analysis of visual data, see Ball & Smith [12, ch. 2] and Bell [23].) Content analysis of architectural diagrams, then, is not such a leap. At a basic level, we can segment and code diagrammatic elements—boxes, lines, clouds, whatever—in much the same way that we can segment and code phrases and paragraphs. But although there's no theoretical barrier to including architectural diagrams in our content analysis, the heterogeneity of our data does create practical challenges. A coding unit in this analysis is something much more complex and multifaceted than a coding unit in a typical content analysis. While in a typical content analysis a coding unit is simply a phrase or a passage, here a coding unit is an aggregation of words, phrases, and passages (occurring in both transcribed speech and written architectural documentation) as well as diagrammatic elements. This complicates segmentation, and it also makes coding itself a much knottier undertaking, since properly categorizing a single coding unit now requires consideration of textual and diagrammatic elements that are spread throughout the material.

5.3.6 Pilot phase

In a qualitative content analysis, a pilot phase allows any problems with the coding frame or the segmentation of the text to be ironed out before the main analysis. After completing a draft of the coding frame and finishing the definition of the coding units, I applied the preliminary coding frame to the segmented text, noting any points of difficulty or confusion. I then made minor changes to the coding guide as appropriate: adjusting boundaries between coding units, clarifying descriptions of categories in the coding frame, and so on. The coding frame that appears in appendix B is the final version, which I used for the main analysis.

Another outcome of the pilot phase was the definition of *context units* to define the scope of the context to be considered when coding a given unit. In the case of content analysis 1, I defined a context unit to consist of the paragraphs immediately surrounding a coding unit. In the case of content analysis 2, I defined the context unit to encompass the entirety of the source material, since occurrences a single coding unit could be spread throughout the data, as explained in section 5.3.5.

5.3.7 Main analysis phase

In the main analysis phase, I categorized each coding unit in accordance with the coding guide I had developed. I carried out the coding for content analysis 1 and content analysis 2 on separate occasions—again treating them as two distinct content analyses even though they are part of a single case study.

I conducted two passes of coding: one pass to identify the codes, and a second to permit evaluation of reliability. I allowed over two weeks to elapse between the passes so that I would not remember the initially assigned categories on the second pass, following a recommendation given by Schreier [195, p. 146]. Once the two passes had been completed, I reconciled the results. For each discrepancy that existed between the initial coding and the recoding, I carefully examined the coding unit, the surrounding context, and the full descriptions of both categories (the one assigned in the initial coding and the one assigned in the recoding) in the coding guide, then chose whichever of the two was most appropriate. This final categorization served the basis for the subsequent interpretation of the results as well as (in the case of content analysis 2) the modeling phase.

5.4 Evolution model construction

The final analytical phase involved the construction of a (partial) evolution model based on the results of content analysis 2. This construction proceeded in stages. First, I constructed initial and target architectures, directly using the results of content analysis 2. Second, I specified a number of evolution operators relevant to the evolution, including several that had been specifically mentioned in the data. Third, I specified the constraints that had been identified in content analysis 2. Finally, I specified evaluation functions for the evolution concerns that had been identified in the content analysis. I will now describe each of these steps in detail.

5.4.1 Initial and target architectures

As section 2.1 explained, our approach is not tied to any particular modeling language or tool. Thus, the first step in constructing an evolution model in this case study was selecting a modeling language. I selected the Acme architecture description language [95] as the modeling language for the case study. Acme was appealing for this purpose because it is designed specifically for systematic, semantically rich representation of software architectures, and it has an extensive set of features for that purpose, such as support for decomposition of architectural elements and definition of rich hierarchies of element types. By contrast, other modeling languages we have used, such as UML, aspire to a much broader set of purposes. Consequently, using such languages with our approach requires care to be taken in establishing explicit conventions for their use.

The content analysis had produced all the significant architectural elements to be modeled and had further categorized them as components, connectors, systems, and so on. In addition, it had determined which of these elements appeared in the initial architecture, and which appeared in the target architecture. With this work done, constructing initial and target architectures in Acme was fairly straightforward.

Of course, the content analysis was not sufficient, on its own, to specify the initial and target architectures fully. It defined and classified all the key elements. But because the interviews and architectural documentation were fairly high-level, they seldom descended to the level of ports and roles, for example. Although the coding guide included a category for ports and roles, most of the ports and roles in the system were never explicitly mentioned. (Acme requires ports and roles to be specified fully. Thus, while in casual conversation one might simply speak of a component *A* being connected to component *B* by a connector *C*, Acme requires the roles of connector *C* to be explicitly defined, and attached to ports on components *A* and *B* that are also explicitly defined.) Similarly, we did not attempt to identify element types in the content analysis; thus, these had to be introduced in the modeling phase.

However, these low-level decisions were all fairly straightforward and inconsequential. All the major decisions—what the major components of the system were, how they should be connected, how the system as a whole was structured—had already been made via the content analysis. At least subjectively, then, we can say that the content analysis succeeded in systematizing and formalizing the major decisions involved in architecture representation, even though it did not itself amount to a comprehensive representation of the architecture.

The initial and target architecture of the system are shown in figures 12 and 13. Not shown in these figures are system substructures modeled using Acme representations. Elements with representations defined are indicated by AcmeStudio's symbol. For the most part, the finer details of the point-of-sale evolution are not important here, but it will be useful to have a passing acquaintance with the system, so I now provide a brief explanation.

Figures 12 and 13 are each divided into several major groups of elements, represented by dashed rectangles, which correspond to the groupings identified in the content analysis. The core point-of-sale system is within the warehouse. The POS element itself does not appear in these diagrams, because it is inside the controller components. The main goal of the point-of-sale evolution is to replace the legacy component at the core of the point-of-sale system, an off-the-shelf point-of-sale package, with another, more modern off-the-shelf package.

At the same time, however, a number of other things are changing in systems with which the point-of-sale system integrates, and these changes will affect the operation

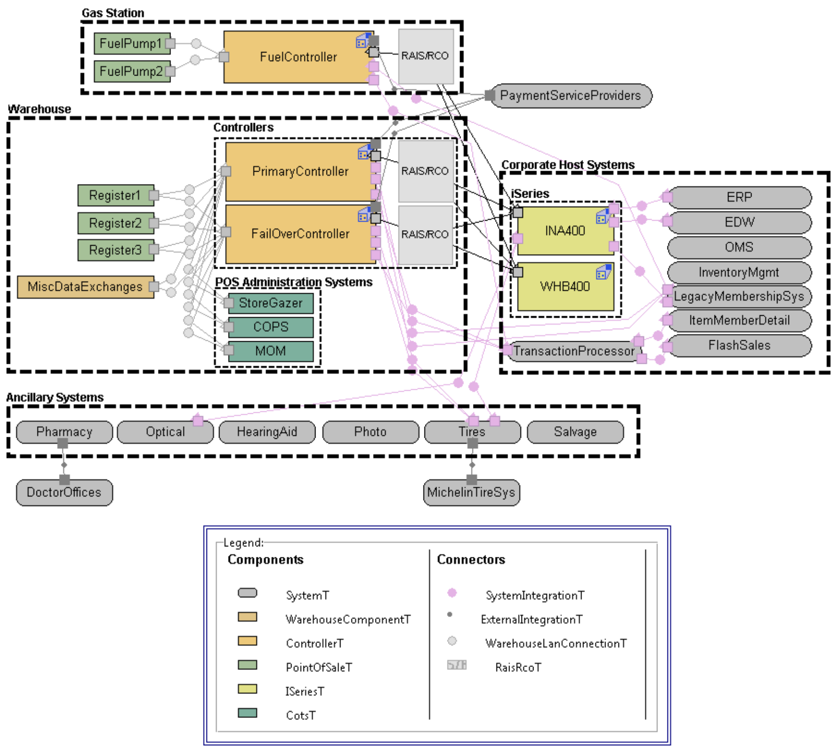


Figure 12. *An AcmeStudio representation of the initial architecture of the point-of-sale system, constructed from the content analysis results.*

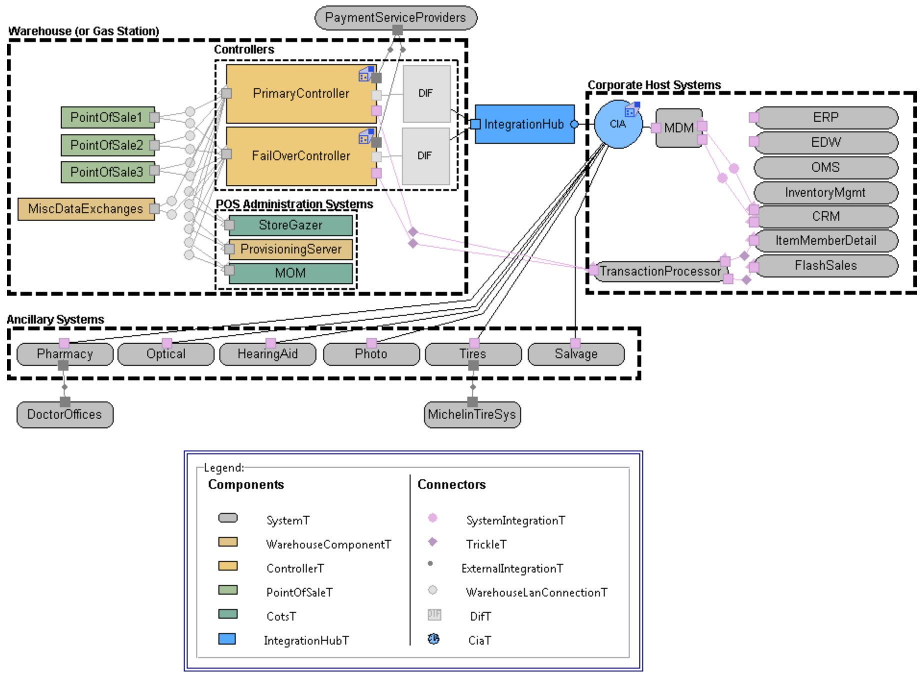


Figure 13. *An AcmeStudio representation of the target architecture of the point-of-sale system, constructed from the content analysis results.*

of the point-of-sale system significantly. For example, a number of new elements are being introduced to facilitate communication among systems, including a new integration architecture element, an integration hub, and a Master Data Management system. These elements will serve to decouple various systems, to unify various communication paradigms in use among systems, and to reduce redundancy. Another significant change is that various legacy systems will eventually be disused (e.g., the mid-range systems that appear in figure 12) or replaced (e.g., the legacy membership system will be replaced with a new CRM system).

Although representing the initial and target architectures based on the results of the content analysis was generally straightforward, there were a few difficulties that did arise. These difficulties may suggest areas for future research or methodological refinement, so I discuss them now:

- The exclusive use of a single (component-and-connector) architectural view was somewhat limiting. The use of multiple view types might have helpfully enriched the model. In particular, supplementing the component-and-connector view with a deployment view might have helped clarify certain aspects of architectural structure. Figures 12 and 13 represent physical boundaries primarily using groupings (e.g., the boundary between the warehouse and the corporate host systems). However, using a deployment view would have permitted us to depict such boundaries more explicitly and precisely. In addition, it would have allowed us to depict which software elements were allocated to which physical devices. This would have eliminated some redundancy. For example, all three of the controllers in figure 12 have the same internal substructure; with a deployment view, this could have been shown with less duplication by depicting individual software elements as being allocated to multiple controllers. In our journal paper [19], we demonstrated how our approach can be used with multiple view types. However, in this case study we used only one view type so as to simplify the content analysis and permit the use of Acme, which does not support multiple architectural views. See section 8.3.4 for a discussion of issues involved in representing an evolution from multiple viewpoints.

- There were several elements about which very little information was provided. For example, the order management system ("OMS" in figures 12 and 13) was mentioned only a couple of times, and then only in passing. I did not receive any information about what it was connected to or about its exact relationship with the point-of-sale system. In fact, it might have been best to omit it from the model, except that I wanted to ensure the model was derived from the content analysis with as few tweaks as possible. There were a number of other elements about which incomplete information was likewise provided. For example, the TransactionProcessor element was discussed in somewhat more detail than the order management system, but still not enough detail to be certain of its correct placement. For example, it might actually be hosted on one of the mid-range computers. The general problem is that the data does not provide an exhaustive description each element. In interviews and documentation, architects tend to focus on elements that are particularly salient, at the expense of more peripheral elements. With more time, I could have remedied this through follow-ups with the study participants. But the difficulty of getting an accurate and comprehensive picture of an architecture also suggests opportunities for future research into methods of architecture elicitation.

- I had some difficulty representing multiplicity. When it was convenient to do so, I used multiple element instances; for example, figure 12 shows three registers and a couple of fuel pumps. (The actual number of registers per warehouse and pumps per gas station is quite a bit higher.) At other times, showing multiplicity was impractical. How can we adequately capture the point that there are hundreds of individual warehouses across the country? We could simply add an annotation—"Warehouse (627 instances)"—but this is a rather crude method. In an intermediate state, there is no good way to show a transitional point in which some warehouses are using a legacy system and some others are using its modernized successor.
- The target architecture was necessarily speculative, because there were certain aspects of its structure that architects weren't yet sure about. For example, a long-term goal is to increase integration between the point-of-sale system and the company's ancillary systems, but it's not yet certain precisely when or how this will happen. In the model in figure 13, I showed these ancillary systems as being fully integrated through the integration architecture, but there are other possibilities that might be just as likely. To some extent this kind of uncertainty is unavoidable, but it also suggests the possibility of capturing the uncertainty in the model itself and reasoning about the trade-offs among different possibilities.
- A subtler difficulty had to do with the way that people describe architectural structure when speaking informally. In formal architecture description, we are very precise about how we describe relationships among elements. Formally, to say that a controller is connected to a mid-range computer is different from saying that the point-of-sale system hosted on that controller is connected to a socket server hosted on the mid-range computer. In everyday language, though, these can amount to the same thing. Thus, when one person describes a connection between the mid-range systems and the warehouse, another person explains that the controller is connected to the mid-range systems, and a third explains that the warehouse has a link to the corporate hosts, they are all referring to the very same connector—even though they are describing it in different ways, and indeed in our content analysis, such utterances were unitized as though they were different entities. In fact, this is not a fictitious example; the content analysis had several different coding units such as "midrange-warehouse," "controller-midrange," "hosts-POS," and several others that all ended up being realized through the same connector in the model.

  One way of understanding such examples is as instances of a very common linguistic phenomenon called synecdoche, in which a part of something refers to the whole or vice versa, as in the phrase "all hands on deck," where "hands" refers to sailors. Similarly, in the example above, we could interpret "warehouse" as synecdochically referring to the controller or vice versa.

There are other ways in which casual language treats architectural structure imprecisely. Sometimes when architects appear to be describing a connection between two components A and B, they are actually describing a communication pathway comprising a series of connectors, in which A and B are the endpoints and the communication goes through a number of intermediaries, so A is connected to C, C to D, and D to B. For example, one architect told me that the CRM system has "an integration point to" the point-of-sale system in the target architecture. In the content analysis, I identified this as a connector between the point-of-sale system and the CRM system. But while reviewing the architectural documentation during the modeling effort, I came to believe that in fact there is no direct link between these two systems in the target architecture; rather, this communication goes through a series of intermediaries: the integration hub, the integration architecture, and the MDM system.

Such complexities suggest a need for further research on how humans talk about architecture and how to elicit precise architectural information from practitioners.

5.4.2 Evolution operators

Because the point-of-sale evolution is a fairly large-scale evolution with many parts and aspects, architects tended to speak of the evolution in terms of its general goals and major stages, rather than the specific individual operations that will be required to carry it out. Nonetheless, the content analysis did identify a few evolution operations at a finer level of granularity, namely:

1. Removing an existing legacy element
2. Connecting the point-of-sale system to a new system
3. Doing in-house customization of an off-the-shelf point-of-sale system
4. Paying the vendor to customize an off-the-shelf point-of-sale system
5. Upgrading the operating system of the warehouse controllers
6. Deploying new point-of-sale software to the warehouse controllers
7. Eliminating the fuel controllers (so that gas stations share warehouse controllers)
8. Replacing nightly polling of transaction logs with a "trickle poll" setup that provides a steady stream of updates throughout the day
9. Installing an off-the-shelf product in the warehouse
10. Modifying an off-the-shelf package to meet the company's needs
11. Replacing legacy communication for file transfers with the managed file transfer capabilities provided by the company's new integration architecture

12. Replacing the data queues used by the socket server with message queues
13. Moving functionalities and responsibilities from a legacy system to the modernized system that replaces it

Obviously, this handful of operators is not sufficient, on its own, to describe all the changes in the point-of-sale evolution—to take us from figure 12 to 13. Because of the scale and complexity of the point-of-sale evolution, a fairly large number of operators would need to be defined to accomplish this. But the operators mentioned in the data are a good basis on which to evaluate our modeling approach. These operators are likely to be no less complex and no less difficult to model than typical evolution operators.

In the full case study report [17, § 4.3.2], I demonstrate how each of these operators can be modeled in our approach. In this dissertation, I show only a few of them.

1. **Removing an existing legacy element** is a straightforward deletion operator, which is not hard to model at all:

```
operator removeElement(e) {
    transformations {
        delete e;
    }
}
```

2. **Connecting the point-of-sale system to a new system** amounts to the creation of a new connector between the point-of-sale system and the new system. To avoid ambiguity, the operator must take as parameters the specific ports to join, rather than just the components.

```
operator integrateWithSystem(clientPort, systemPort) {
    transformations {
        Connector c = create Connector : SystemIntegrationT;
        attach clientPort to c.Client;
        attach systemPort to c.Sys;
    }
    preconditions {
        declaresType(clientPort, SystemCallT) and
        declaresType(systemPort, SystemIntegrationPointT)
    }
}
```

11. **Replacing legacy communication for file transfers with the managed file transfer capabilities provided by the company's new integration architecture** is only slightly more complex than the examples we have seen. In this operator, the legacy connector used to allow the warehouse controllers to interact with the corporate host systems is disused and replaced with a connection to the integration hub.

```
operator disuseLegacyFileTransferCommunication(legacyComm, integrationHub) {
    transformations {
        for (Port p : legacyComm.Ctlr.attachedPorts) {
            remove type p : ControllerLegacyCommPortT;
            add type p : MicrobrokerHookupT;
            Component microbroker = create Component : MicrobrokerT;
            attach p to microbroker.Controller;
            attach integrationHub.IntegrationPoint to microbroker.IntegrationHub;
        }
        delete legacyComm;
    }
    preconditions {
        declaresType(legacyComm, LegacyCommElementT) and
        declaresType(integrationHub, IntegrationHubT) and
        size(legacyComm.Ctlr.attachedPorts) = 1 and
        forall p in legacyComm.Ctlr.attachedPorts |
            declaresType(p, ControllerLegacyCommPortT);
    }
}
```

These operators are remarkable for how straightforward and easy they were to specify. Even the operators that seemed conceptually complex turned out to be simple to define. Although we should be careful of drawing too strong a conclusion from this limited sample, this suggests that the operators that arise in real-world evolution may tend to be fairly simple in many cases, and that operator specification is not as difficult as might be feared.

5.4.3 Constraints

Sixteen constraints were identified by the content analysis. A detailed discussion of each identified constraint appears in the full case study report [17, § 4.3.3]. Here I give a brief summary.

In segmenting and coding the data, I took a fairly inclusive view of what it means for something to be a "constraint." I included in this category everything that took the form of a constraint somehow governing the evolution, without considering whether it was an architecture-level constraint or a lower-level constraint, whether it was a constraint on the evolution as a whole or only particular states, or whether it was amenable to representation via our approach.

As a result, not all of the constraints identified in the content analysis are easily representable as evolution path constraints using our approach. Of the sixteen constraints identified by the content analysis:

- Seven can be straightforwardly represented as evolution path constraints. An example of such a constraint that arose in the content analysis is a constraint that ACE can never be hooked up to the socket server. Such a constraint can be modeled easily by

$$\Box newPosNotConnectedToSocketServer(system),$$

where *newPosNotConnectedToSocketServer* is a unary predicate over architectural states, definable in Acme's architecture constraint language, Armani, by:

```
Design Analysis newPosNotConnectedToSocketServer(s : System) : boolean =
    forall ctlr : Component in s.components |
        forall ctlrRep : Representation in ctlr.representations |
            forall pos : Component in ctlrRep.components |
                forall posRep : Representation in pos.representations |
                    forall c : Component in posRep.components |
                        c.name == "NewPOS" -> !connected(ctlr, WHB400);
```

- Four are very high-level constraints that were not described in sufficient detail to model fully, but are representable in principle. For example, one architect mentioned availability constraints as being particularly significant to the point-of-sale system, noting that some elements of the point-of-sale system are expected to have 99.999% availability.

  Of course, accurately predicting system availability from architectural structure is a challenging problem—one that has received significant research attention in its own right [101]. It would be interesting to study how well our approach could capture architectural reliability models that have appeared in previous research, but that is well beyond the scope of this case study.

  For now, let us take a simple example. Suppose that we were interested solely in the reliability of the controllers in the warehouse. A warehouse has two redundant controllers: a primary and a fail-over. We can annotate each of these two components with a property indicating its expected failure rate (e.g., $failureRate = 0.001$ to indicate that the component will be down 0.1% of the time.). If we assume that the two controllers fail independently, we can calculate the overall failure rate of the system by multiplying the failure rates of the two controllers. (Of course, independence will not hold in practice. If the warehouse loses power, both controllers will fail. But it is a reasonable simplifying assumption for our demonstration here.) The constraint that availability never drops below 99.999% then becomes

$$\Box highReliability(system),$$

  where *highReliability* is defined in Armani by

```
Design Analysis highReliability(s : System) : boolean =
    s.PrimaryController.failRate * s.FailOverController.failRate <= 0.00001;
```

- Three are low-level, chiefly nonarchitectural constraints, but are representable to the extent they implicate architectural considerations. For example, one architect strongly emphasized the importance of minimizing the impact of the system change on its users, noting that the "first and absolute foremost goal is to create almost a zero-impact change in our warehouses." This is a difficult constraint to capture using our methods, because in large part, it is not architectural. That is, avoiding disruption to users has a great deal to do with the specifics of exactly how the change is carried out, and details like user interface elements, and not much to do with architectural structure. In fact, the same architect made a similar point later in that interview, noting that carrying out the evolution would require attention to minute, nonarchitectural details such as the color of the labels on the keys of the cash registers:

> The things that boggle my mind we talk about in this project—well, we need to know what color code to put on the key cap. I say, oh, yeah, I never really thought about that. But they're training people that identify sections of that keyboard. So you learn a little bit more every time you turn around. It's kind of interesting. It's like, ugh, these darn labels—I forgot all about them! [*laughs*] And really, is that something the architect is concerned about? Probably not.

But to the extent that this "zero-impact change" constraint does implicate architectural considerations, we should be able to model it. For example, a simple structural constraint like "The number of cash registers in the warehouse remains constant throughout the evolution" is easy to model.

- One is representable as an operator precondition: a constraint that any new processes or equipment in the point-of-sale system would require recertification with the electronic-payment-services vendor. The most straightforward way to model this constraint is as a precondition on the operator that would modify the certified equipment. For example, if we have an operator such as *modifyCashRegisters*, it should have as a precondition that the *recertifyEps* operator has already happened to ensure that the modifications to the cash registers have been approved.
- One is representable only with significant modifications to the model: a constraint that the legacy warehouse communication elements that facilitate communication to the corporate hosts shouldn't be removed in *any* warehouse until *all* warehouses are on the new point-of-sale package. Because we have chosen to model only a single illustrative warehouse, we can't represent constraints that require consideration of differences among different warehouses. However, if we enriched the architectural model with multiple warehouses, this would be straightforward to model.

The exact tallies in this list are not important. In many cases, there was some degree of choice in representing the constraints. Ordering constraints, for example, can typically be represented either as evolution path constraints or as preconditions.

What is important is the overall result: all the architectural constraints captured in the content analysis are representable using our approach, albeit with varying degrees of difficulty. We have seen some specific areas where our constraint language might be able to benefit from further enrichment, such as support for reasoning directly about the transitions in an evolution graph in addition to the states. But we did not encounter any architecture evolution constraints that could not be modeled at all.

5.4.4 Evaluation functions

The content analysis identified seven dimensions of concern: total cost, total effort, return on investment, performance, implementation details, effort spent on modifying legacy technology, and flexibility. Most of these can be modeled as evaluation functions with little trouble. In our full case study report [17, § 4.3.4], we examine each of these seven dimensions of concern in detail. Here I provide a brief summary.

Of the seven dimensions of concern identified in the content analysis, five can be modeled as evaluation functions with little trouble. One such concern is total cost. In our approach, cost can be modeled as a property of operators. Each operator can be assigned a property describing its estimated cost in dollars. For example, if we know that the *replaceDataQueues* operator costs 500 dollars to carry out, we can add a property to the definition of the operator:

```
operator replaceDataQueues(subsystem) {
    transformations { ... }
    analysis { "costInDollars": 500 }
}
```

Of course, in some situations, cost may be more complex—depending, for example, on the operator parameters—but this is the basic idea. Then, an operator analysis can simply add up all the operators within an evolution path to get an estimate of the overall cost of the path.

```
function analyzeCost(states, transitions) {
    var total = 0;
    transitions.forEach(function (transition) {
        transition.operators.forEach(function (operator) {
            total += operator.analysis.costInDollars;
        });
    });
    return total;
}
```

The only difficult task is accurately estimating the costs of the operators. However, this is outside the scope of my research. There is a very large body of work on cost estimation in software engineering, so we may simply assume that the costs of the operators can be determined through traditional methods.

Two of the dimensions of concern are not so easy to model. One concern identified by the content analysis pertains to low-level implementation details not relevant to an architectural analysis. However, the other, flexibility, is a genuine architectural concern that cannot be easily analyzed using our approach. System flexibility is quite difficult to estimate based on an architectural model. There is an existing body of research on architectural analysis of the flexibility, evolvability, or changeability of software systems (see section 7.3).

I did not attempt to implement any architectural analyses for flexibility as part of this case study. To the extent that flexibility is determinable from standard architectural models, it is analyzable using our approach. However, many existing methods for architectural analysis of flexibility are not based chiefly on an architectural model, but instead rely heavily on procedures such as definition of change scenarios [25; 127] or interviews with stakeholders [70]. Such methods lie well beyond the scope of the kind of analysis our approach aims to support.

5.4.5 Evolution styles

Before concluding this section on evolution model construction, let us return to the topic of evolution styles, introduced in section 2.6, and examine how it could be applied to an evolution like the one examined in this case study. Recall that an evolution style is a way of encapsulating a family of operators, constraints, and evaluation functions relevant to an evolution domain. This case study did not set out to examine the applicability of the concept of evolution styles, and no evolution styles were constructed during the course of the case study. Nonetheless, it may be instructive to discuss how an evolution style relevant to the point-of-sale evolution might be defined, incorporating some of the evolution elements described in the preceding sections.

The first necessary element of an evolution style is a description of the domain to which it is relevant. In defining a domain relevant to the point-of-sale evolution, there are a few different options that might serve as a reasonable basis for an evolution style. We could, for example, choose to define an evolution style for the domain of *point-of-sale systems*. This might be feasible, but it is a bit too specific for the purpose of this discussion. Architecturally speaking, there is little that distinguishes a point-of-sale system from other kinds of systems that are architected in a similar fashion (and conversely, different point-of-sale systems might have very different architectures). A better choice might be to think of our domain of evolution as *technology upgrades within the context of a hub-and-spoke architecture, in which some corporate host system serves a large number of remote, distributed systems*. Such a domain is broad enough to encompass a very large class of systems; just about any large company will have many systems fitting that description. But it is also specific enough that we can define useful and concrete evolution elements relevant to the domain.

Indeed, many of the evolution elements defined in the preceding sections or in the full case study report [17, § 4.3] would be suitable for inclusion in such an evolution style, including operators such as upgrading a system or customizing an off-the-shelf package or introducing an integration bus, constraints enforcing rules governing how and when these operators may be applied, and evaluation functions for concerns such as cost and performance. Of course, to define an evolution style fully, we would need to define a large number of operators, sufficient to completely specify an evolution instance in the domain. Additional constraints and evaluation functions would likely be needed as well. But once the evolution style is defined, we could apply it to a broad class of evolutions, such that the cost of defining the evolution style may be amortized over a large number of reuses.

There are a couple of unanswered research questions that this examples reveals. First, it suggests a need for some notion of specialization or extension of evolution styles. The evolution domain defined in this example—technology upgrades in a corporate hub-and-spoke system—might be a specialization of a style for technology upgrades generally, and might in turn be specialized into a more specific substyle for technology upgrades of point-of-sale systems. Support for defining evolution styles by inheriting elements from other evolution styles would facilitate reuse and make it more practical to define and relate evolution styles. A second point is that many evaluation functions (and perhaps other elements) are not really domain-specific. Our evaluation function for cost is in no way specific to point-of-sale evolutions or hub-and-spoke evolutions; it simply adds up the costs defined by the operators. It therefore doesn't really make sense to define cost properties and cost evaluations as part of a conventional evolution style; instead, we might imagine defining some kind of mix-in style for cost analysis that could somehow be combined with other, conventional evolution styles. We will return to these issues in section 8.3.1, which discusses the need for future work on evolution styles.

5.5 Findings

In section 5.3, I discussed the two content analyses used in this case study: one (content analysis 2) feeding into an evolution modeling phase and one (content analysis 1) to be interpreted directly by conventional means. In section 5.4, I discussed how the modeling phase following content analysis 2 was carried out and what its findings were. In this section, I examine the findings of content analysis 1. In section 5.6, I will synthesize these findings and explain how they address the research questions defined at the outset of the study.

The following subsections correspond to the top-level categories of content analysis 1: "Motives for evolution" (section 5.5.1), "Causes of problems" (section 5.5.2), "Consequences" (section 5.5.3), "Challenges" (section 5.5.4), and "Approaches" (section 5.5.5). It may be helpful to refer to the coding guide in appendix B.1 while reading the following findings.

5.5.1 Motives for evolution

The first topic that the content analysis examined was the impetuses that motivate evolution—the catalysts that drive an organization to carry out major changes to its software systems. Of course, there is a great deal of previous research on reasons for software changes. One of the first influential taxonomies of software change was the one proposed by Swanson [213] in 1976, which identified three main kinds of software maintenance: (1) *corrective maintenance*, designed to address failures of a software system; (2) *adaptive maintenance*, designed to respond to changes in the environment; and (3) *perfective maintenance*, designed to improve a software system that is already working adequately. Swanson's taxonomy was so influential that it was incorporated, many years later, into ISO/IEC 14764 [109, § 6.2], an international standard on software maintenance, with the addition of a fourth type: *preventive* *maintenance*. For a more modern ontology of software maintenance, see Chapin et al. [48].

Very little research has been done on motivations for *architectural* evolution specifically. Of course, many of the motivations for low-level software maintenance can also be motivations for architecture-level software evolution and vice versa, but there are likely some important differences, which previous work has rarely examined. Williams & Carver [222] developed a taxonomy called the Software Architecture Change Characterization Scheme, which categorizes architectural changes along a number of dimensions including a change's motivation, source, importance, and so on. Their taxonomy incorporates only two classes of motivation for architecture changes: changes motivated by the need for an *enhancement*, and changes in response to a *defect* [222, § 5.2.1]. Besides being reductive in the extreme, this dichotomy has another problem: Williams & Carver's taxonomy is based on a literature review of work on software maintenance generally (such as the aforementioned work by Swanson and Chapin) rather than any research specifically focusing on architecture-level change. Thus, while it purports to be a taxonomy of architectural change, it is not actually based on architecture research. Similarly, Jamshidi et al. [112, table 6], in their classification framework for "architecture-centric software evolution," borrow the software maintenance typology from ISO/IEC 14764 for their "Need for Evolution" dimension.

I am aware of one empirical study that has examined causes of architecture-level evolution specifically. Ozkaya et al. [178, § 4.2.1], in an interview-based survey of nine software architecture projects, collected data on the causes of the evolutions that the survey participants described. Responses included new features, market change, technology obsolescence, and scalability.

Clearly, more research investigating motivations for architectural change is needed.

Of course, the objective of this case study was by no means to develop a general taxonomy of motivations for architecture evolution. Data from a single case study at a single organization would provide insufficient evidence to support such a taxonomy even were it my wish to construct one. But what this data can do is help inform us as to how practicing architects in a real software organization think about and describe the causes of evolution, and it can also illuminate the rest of the case study by properly setting the context.

The interviewees I spoke with mentioned a variety of motivations for evolution, which I grouped into seven categories. These are listed in table 3.

Table 3 illustrates the basic format that all the frequency tallies from the content analysis will use. The subcategories of the relevant category from the coding frame are listed, along with the frequencies with which they appeared in the coded data. The frequency of a category can be reasonably measured in any of three ways in this study: by counting the number of individual coding units to which the category is assigned, by counting the number of interviews in which the category occurred at all, or by counting the number of participants who had any statements to which the category was assigned. (There is not a one-to-one correspondence between interviews and participants because some participants took part in multiple interviews, and in some interviews two participants were present.) None of these measures is

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Add features	12	4	3
Improve interoperability	9	3	2
Modernize technology	5	4	3
Keep apace of business needs	3	3	3
Improve reliability	3	2	2
Improve flexibility	3	1	1
Improve performance	2	2	2

Table 3. *Stated motivations for architecture evolution*

perfect. Probably the most customary choice would be the number of interviews, since content analysts most often count their results with respect to the unit of analysis, which in this case is an interview. However, the divisions between interviews are not particularly meaningful here. For example, in one case an interview was ended on Monday afternoon so that the participant could head home at the end of the workday, then resumed on Tuesday; this was treated as two separate interview sessions. So a more natural measure might be the number of participants, rather than the number of interviews. Both of these measures, though, are quite coarse, since the total number of participants was rather low. A more fine-grained measure is the number of coding units, but this has problems too. The coding units in this study varied dramatically in length, ranging from a short phrase to multiple paragraphs, so different coding units may not carry equal weight. Although none of these measures is perfect, in concert they give a very good sense of how often the categories occurred.

Several of the categories that appear in table 3—improving flexibility, improving performance, improving reliability, and improving interoperability—fall under the general heading of improving architectural qualities. Thus, if I were to summarize these results in a sentence, I would say that most of the evolutions that architects described arose from one of three general motivations: adding new features, modernizing old technology, or improving a system's architectural qualities.

The full case study report describes in detail each category that appears in table 3 and interprets it with respect to the interview data I collected. For space reasons, I do not reproduce this discussion here. Before concluding, however, is is important to note that these results—and especially the relative frequencies with which the categories occurred—are probably somewhat specific to Costco. Costco is a decades-old company with a mature IT organization—one that is at present undergoing significant growth. Thus, it is not surprising that much of the evolution effort there is focused on modernizing obsolescent systems and ensuring that the architecture can meet the business's performance and reliability needs, for example. Were we to repeat this case study in a different kind of company—a five-year-old start-up, for example—the results would likely be different. A small, young organization would probably be much less driven by issues of technology modernization and scale, and perhaps more driven by feature development or other kinds of evolution. (Evolution in general would likely also constitute a lower proportion of software development activities, with greenfield development of new software systems being more prominent.) Thus, in generalizing these results, and those of the following sections, it is important that we keep in mind what special qualities of the case under study may have influenced the results, and to scope our generalization accordingly. I will return to the issues of generalization and external validity in section 5.6.3.

Finally, it should be emphasized that the fact that certain causes don't appear in table 3 does not mean that such causes are not important. For example, the fact that technical debt, or the architectural messiness of legacy systems, does not appear here doesn't mean that that isn't an important factor in evolution for the company. On the contrary, it certainly is. However, the participants in the study did not generally speak in those terms in their impromptu explanations of reasons for evolution—or to the extent that they did, those remarks were categorized under "Evolution motives: Modernize technology" and thus grouped in with other considerations pertaining to the need to modernize technology. Thus, care should be taken in drawing negative conclusions from these results—conclusions that unnamed causes are *not* important.

The categories that appear in table 3 are those that naturally emerged from the data. (Recall that the definition of the subcategories in content analysis 1 was almost entirely data-driven, or inductive, not concept-driven, or deductive.) The results here thus do several things: they highlight some important classes of evolution motives that demonstrably do occur in a real-world organization, they allow comparison of the relative frequency with which those motives were reported, and they show how architects in the case study described the causes of evolution. But they don't permit us to infer anything about the frequency or importance of causes that were not explicitly mentioned.

5.5.2 Causes of problems

The second top-level category of the content analysis was causes of problems—circumstances that architects described as leading to specific negative outcomes during the course of an evolution project. (I also coded the consequences of such circumstances; see section 5.5.3.)

Of all the major categories, "Causes of problems" was the least frequent, and "Consequences" was the second least frequent. This means that participants spoke less about problems than they did about motives, challenges, and approaches. There are a few possible reasons for this. One possibility is that participants were reticent to speak about the company's failures; it would be understandable that architects might want to put their best foot forward in representing the company to an outsider, especially a researcher who was going to make the results of his study public. There may be an element of truth to this. However, my impression was that the people I spoke with were generally quite candid with me. Many of them were quite frank in discussing disagreements that they had had with other architects, or cultural factors that they believed were problematic. Many of them also expressed a sincere desire

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Cultural inertia	4	2	2
Architects spread across the organization	2	1	1
Addressing too many problems up front	2	1	1
Lack of experience	1	1	1
Forking an off-the-shelf package	1	1	1
Ill-timed changes	1	1	1

Table 4. *Stated causes of evolution problems*

to facilitate the case study and to help me obtain the best results possible. It seems unlikely that problems were deliberately concealed.

That said, it is certainly possible that participants phrased their answers diplomatically to avoid casting their employer in an unduly negative light. For example, if an architect believed that decision X had been a mistake, he might not directly say, "We made a mistake with decision X and ran into trouble because of that." Instead, he might say, "Making good decisions about X has been particularly challenging for us." But if he phrased it this way, the statement would be coded as a challenge instead of as a cause of a problem.

This brings us to another factor that could have contributed to the low number of coding units: the coding frame. The coding frame provided specific guidance for distinguishing between causes of problems and challenges: only if a circumstance caused a specifically articulated negative outcome should it be categorized under "Causes of problems;" otherwise, it should be categorized under "Challenges." Perhaps this criterion was too constraining, and greater liberties should have been taken in reading between the lines to distinguish between general challenges and causes of specific problems. Alternatively, perhaps these two categories should have been merged to avoid the issue entirely.

A final possibility is that there genuinely have not been many poor decisions that resulted in adverse consequences. Certainly, the company has a mature IT organization with many experienced, thoughtful, and skilled architects. Perhaps they have successfully averted or mitigated issues that might otherwise have caused significant problems.

Because there were few coding units identified as causes of problems, each subcategory had only a handful of occurrences—and in some cases, only one. The frequencies are given in table 4. These categories are discussed and explained in the full case study report [17, § 5.2]. For space reasons, I do not reproduce this discussion here.

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Wasted effort	6	4	4
Delays in evolving	2	2	2
Limited upgradability	1	1	1
Lost sales	1	1	1

Table 5. *Stated consequences of evolution problems*

5.5.3 Consequences

"Consequences" is the sister category to "Causes of problems." It describes the bad outcomes that resulted from issues such as those discussed in section 5.5.2. The frequencies from the content analysis appear in table 5. For further discussion of this category, see the full case study report [17, § 5.3].

5.5.4 Challenges

An interview-based case study provides a unique opportunity to learn about the challenges that real-world architects face. Software architecture research, including our research, is often driven by researchers' beliefs about what will help practicing architects. These beliefs are often founded on our subjective impressions, generalizations from our own experiences, and informal conversations with practitioners. However, it is important to ground these beliefs with real empirical data whenever possible, and one of the best ways to do so reliably is to ask architects about the challenges they face in the context of a research interview.

"Challenges" should be understood broadly here. I collected data on all kinds of challenges that architects face in evolving systems—not just the kinds of challenges that appear to be addressable through new tools or approaches. Encouraging architects to speak freely about the challenges that they face, rather than simply asking them about the narrow classes of challenges that our research is designed to address, gives us the broadest and least biased picture of architects' needs, and helps us to accurately understand the role that approaches like ours can play in addressing a realistic subset of the challenges that architects face.

Frequencies appear in table 6. The most common code was "Communication, coordination, and integration challenges," an expansively defined category that includes a broad range of difficulties, including challenges of communication among people, challenges in coordinating efforts, and challenges in integrating systems. When developing the coding frame, I chose to group these somewhat disparate topics together under a single category because issues of organizational communication are often closely tied to issues of software integration. With 22 occurrences, this category was extremely common—the second most frequently applied category in the entire coding frame. Broadly speaking, this suggests that participants view

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Communication, coordination, and integration challenges	22	7	5
Dealing with rapid change and anticipating the future	13	6	4
Business justification	9	6	5
Dealing with legacy systems	9	5	5
Scalability, reliability, and performance	8	2	3
Managing the expectations and experience of users and stakeholders	7	4	5
Managing people	7	3	3
Inadequate guidance and tool support	6	4	4
Cultural challenges	5	3	3
Lack of expertise or experience	5	3	3
Divergent understandings of architecture	4	2	2
Managing scope and cost	3	3	3
Dealing with surprises	3	2	2

Table 6. *Stated challenges of architecture evolution*

communication and coordination challenges as a very important consideration in architecture evolution efforts.

One significant theme that emerged particularly often was integration issues among simultaneously evolving systems. Because so many different systems are evolving as part of the company's overall modernization effort, integration issues among those systems arise continually. Architects have to understand not only the future state of their own system, but also that of the other systems with which their system interoperates. One architect working on the point-of-sale evolution studied in section 5.4 explained:

> It's not enough to say I'm going to upgrade my point-of-sale system, which is a huge and daunting task in and of itself here at Costco. When you talk about all of the dependencies and all of the different components that are going to be interacting with point-of-sale, now you've got a unfathomable challenge in front of you, and it really takes a lot of focus to keep from getting into trouble.

After communication and coordination issues, the next most frequently occurring category was “Dealing with rapid change and anticipating the future.” This, too, is an umbrella category that encompasses a few different meanings: planning ahead in the face of rapid change, seeing the long-term ramifications of decisions, estimating future effort, adapting industry advances to the company, and realizing a long-term vision. What all these topics have in common is that they describe challenges involved in making good decisions in a rapidly changing environment. The high frequency with which this category appeared in the data suggests that this is viewed as a particularly important challenge. As one participant explained:

> You really have to have that ability to shift gears and change directions quickly, because the technology landscape changes so fast, and when you’re on a project that’s going to run three to five years, changes are inevitable.

The third most frequent category of challenge was “Business justification,” which captures challenges in justifying architectural decisions in business terms, ensuring that software systems support business goals, understanding business needs, and so on. One participant characterized executive-level support for architecture as the biggest single issue that the industry faces with respect to architecture evolution, while emphasizing that this is not such a big problem at Costco:

> I think in terms of architecture, what I’ve seen the biggest challenges are executive buy-in on the architecture function in general, because when you throw architects in at the beginning (goodness, I hope it’s at the beginning) of a project, you’re adding time to it, and when you’ve got a request from the business, or you’re comped on time to market, then there’s not a whole lot of effort that you want to put into initial architecture. We’ve been fortunate that we’ve got CIO buy-in of what we’re doing. But I think that’s probably the largest obstacle, and when I meet with other people who run EA and solutions architectures groups from other companies, they always say that that’s the largest challenge they face.

“Dealing with legacy systems” is a reasonably self-explanatory category. This was explained to me particularly vividly in an interview with a pair of architects:

> **Participant 1:** One of the sad things that Costco came to realize is that the applications that were built [. . .] here over time—over the twenty-five or thirty years or so that the mid-range systems were here—really grew organically, and there wasn’t comprehensive architectural thinking going on in the early days, to the point where you had a big ball of string. And now it’s so brittle that the simplest changes are taking months and months, and it’s just not really good.
>
> **Participant 2:** Part of the problem is understanding the existing architecture that exists, even if it wasn’t architected by design.

> **P1:** Yeah, and I would say that's probably optimistic—understanding the architecture. Really there was no architecture. [. . .] We got an analysis tool that we put onto our system and we had it do a drawing of all of the components and the relationships, and honestly it looked like a ball of yarn—the big red dots all around the outside with lines going everywhere. We said, well, that's the problem right there.

This can be viewed quite naturally as an example of technical debt. In failing to manage the overall design of its early systems as they were built, maintained, and evolved over the years, the company has been accruing a kind of architectural debt—debt which is now coming due as the company struggles to make major changes to these systems.

For further discussion of these categories as well as explanations of the other categories of challenges, see our case study report [17, § 5.4].

5.5.5 Approaches

This final section of findings from content analysis 1 answers the question: what approaches do architects in a real-world software organization currently use to manage major evolution efforts?

At the beginning of the study, I conjectured that architects currently have few practical approaches for managing major evolutions, and that I would find that architects therefore have little to say about the techniques they use to plan and carry out evolution. But in fact "Approaches" turned out to be the most frequently occurring of all the major categories. Not only that, but the coding units in this category were longer, on average, than the coding units in any of the others. Thus, architects frequently and at great length about the methods they use to manage evolution.

As with the other categories, the topic of this category was construed quite broadly for coding purposes. That is, the "Approaches" category includes not only concrete techniques such as tools and formal methods, but also very informal strategies and rules of thumb. Thus, the mere fact that the "Approaches" category occurred frequently does not, by itself, contradict the conjecture that architects lack formal techniques for managing software evolution. That question can be addressed only once we have examined the subcategories, which we will now proceed to do. The frequencies are given in table 7.

Just as "Communication, coordination, and integration challenges" was the most frequently occurring subcategory of "Challenges," so is "Communication and coordination practices" the most frequent subcategory of "Approaches"—indeed, the most frequently occurring subcategory in the entire coding frame. This suggests that communication and coordination issues are tremendously important in managing architecture evolution. As one architect explained:

> A lot of this is just doing what works for you. It's really more about communication than architecting systems. That collaboration aspect, I think, is absolutely paramount to the success of an architect. You have

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Communication and coordination practices	25	7	6
Organizational strategies and structures	17	5	5
Phased development	15	4	3
Drawing from expert sources	13	5	4
Rules of thumb and informal strategies	13	5	4
Tools	12	5	5
Anticipating the future	8	5	4
Prototypes, pilots, etc.	8	4	4
Training	8	4	4
Business architecture	8	3	3
Formal approaches	6	4	4
Experience and intuition	6	3	2
Process	4	3	3
Industry practices	3	3	3
Considering alternatives	1	1	1

Table 7. *Stated approaches for architecture evolution*

to talk to lots of people all the time. The drawings don't have to be that precise, as long as they communicate the information.

Participants mentioned a variety of communication and coordination practices that can facilitate architecture evolution, including prioritization of projects, appropriate use of architectural documentation, coordination on dates of delivery for internal services, organizational standardization on certain technologies, alignment of parallel efforts, providing guidance on best practices within the organization, understanding political implications of architectural decisions, and maintaining a feedback loop between architects and developers.

The second subcategory, "Organizational strategies and structures," captures architects' explanations of the organizational structure of the IT department and how it supports architectural efforts. I have already described the organizational structure in section 5.1.2.

There were 15 occurrences of the category "Phased development," which captures situations in which architects described breaking projects down into manageable phases or stages. As one architect told me, planning a large, complex evolution in phases is often a practical necessity: "you can't just Big Bang something like that." This category is significant because it describes the same basic approach that we have taken in our research: breaking down a large evolution into smaller steps that can be more easily understood and managed (or conversely, building up small operations into a large evolution). The high frequency with which architects described this technique in the interviews suggests that in this respect, architects are already using the kind of stage-based reasoning that our approach is based on. This is an encouraging sign for the realism and practicality of our approach.

The remaining categories are discussed in detail in the case study report [17, § 5.5]. Here it suffices to conclude that architects currently have a variety of approaches that they use to manage architecture evolution, but many of these approaches are highly informal or consist of general strategies rather than specific methods. The architects I spoke with do not have tools of the sort that our research aims to develop: tools for developing and evaluating end-to-end plans for rearchitecting a software system.

5.6 Conclusions

5.6.1 Answers to the research questions

I now return to the case study research questions articulated at the beginning of this chapter and document explicitly how they have been answered.

1. **How do practicing architects in a real-world software organization plan and reason about evolution?** This question was addressed by content analysis 1, particularly the "Approaches" category of the content analysis (and to a lesser extent the "Evolution motives" category, which describes the significant catalysts that architects mentioned as driving evolution). To summarize, we found that architects have a remarkably wide range of strategies and approaches that they use to manage architecture evolution. The most prominent category of approaches in the interview data was communication and coordination practices, suggesting that issues of communication and coordination are very important in managing architecture evolution. Other particularly frequently mentioned approaches for managing architecture evolution included the use of organizational strategies and structures, breaking down software development into manageable phases, taking advantage of expert sources such as consultants and books, and using various rules of thumb and informal strategies.
2. **What difficulties do practicing architects face in planning and carrying out evolution?** This research question was also addressed by content analysis 1—particularly the "Challenge" category, as well as the "Causes of problems" and "Consequences" categories. By far the most commonly mentioned category of challenges was "Communication, coordination, and integration challenges." This corresponds neatly to the result mentioned above that "Communication and coordination practices" was the most frequent subcategory of "Approaches." This reinforces the conclusion that communication and coordination issues are extraordinarily important in managing architecture evolution in practice. Specific challenges mentioned within this subcategory include documentation challenges, challenges of architecture conformance, and challenges in coordinating with an off-site development team, among others.

Other significant classes of challenges identified by the content analysis include anticipating future developments, justifying decisions in business terms, dealing with legacy systems, assuring architectural qualities such as scalability and reliability, and managing stakeholder expectations.

Understanding the challenges that architects face helps us to position our research with respect to the current state of architectural practice. Our work has at least a small role to play in many of these challenges. For example, one of the goals of our work is to support reasoning about integration activities during the course of an evolution, which might help forestall certain integration problems. But clearly there are some categories, such as “Managing people” and “Cultural challenges,” to which our work is irrelevant.

3. **How well can our modeling framework capture the concerns that arise in a real-world architecture evolution?** This research question was addressed by content analysis 2 and the evolution model that was constructed based on it. The construction of the evolution model is described in detail in section 5.4. The great majority of the operators, constraints, and evaluation functions identified by the content analysis could be modeled using our approach, although a few of them were not suitable for modeling, often because they were too low-level (pertaining to implementation details rather than architecture) or too high-level (encompassing a broad range of specific considerations that were not understood in sufficient detail to model).

These results are generally encouraging. However, understanding their significance requires us to assess their reliability and validity.

5.6.2 Reliability

There are two broad quality dimensions that are of paramount importance in qualitative research: validity and reliability. An instrument is said to be *valid* to the extent that it captures what it sets out to capture and *reliable* to the extent that it yields data that is free of error. In this section, I evaluate and discuss the reliability of this case study. In the following section, I will consider validity.

The most well-known treatment of content analysis reliability is that of Krippendorff [122], who distinguishes among three kinds of reliability:

- **Stability**, the degree to which repeated applications of the method will produce the same result
- **Reproducibility**, the degree to which application of the method by other analysts working under different conditions would yield the same result
- **Accuracy**, the degree to which a method conforms to a standard

These are listed in order of increasing strength; reproducibility is a stronger notion of reliability than stability is, and accuracy is the strongest of the three. However, accuracy is only occasionally relevant in content analysis, because it presumes that there is some preexisting gold standard to use as a benchmark, such as judgments by a panel of experienced content analysts. In many content analyses, especially those exploring new domains or topics, there is no such gold standard to use. In addition, as Krippendorff [122, p. 272] explains, accuracy is a problematic concept for content analysis even in principle: “Because interpretations can be compared only with other interpretations, attempts to measure accuracy presuppose the privileging of some interpretations over others, and this puts any claims regarding precision or accuracy on epistemologically shaky grounds.” Thus, the two main forms of reliability which are normally relevant to content analysis are stability and reproducibility.

Let us begin with stability. The most direct way of assessing the stability of an instrument is through the use of a test-retest procedure, in which the instrument is reapplied to the same data to see if the same results emerge. In fact, stability is sometimes called “test-retest reliability” or simply “repeatability.” I incorporated a test-retest procedure into the design of the content analysis. For each of the two content analysis, I conducted a second round of coding at least 17 days after the initial coding. (As mentioned in section 5.3.7, Schreier [195, p. 146] recommends that at least 10–14 days elapse between successive codings by the same researcher.) Once two rounds of coding have been completed, the results can be compared to evaluate the degree to which they agree. This provides a measure of intrarater agreement (also known by other similar names such as intracoder reliability, intra-observer variability, etc.).

Intrarater agreement can be quantified using the same metrics that are used to measure interrater agreement in studies with multiple coders. There are a large number of such metrics [84, ch. 18; 184], but there are four that are most popular in content analysis: percent agreement, Scott’s π, Cohen’s κ, and Krippendorff’s α.

Percent agreement is the simplest and most obvious metric for quantifying intrarater or interrater agreement. It is exactly what it sounds like: the percentage of coding units that were categorized the same way in both codings. For example, in our content analysis 1, there were 306 coding units total across the eight interviews, of which 280 were categorized the same way in the initial coding and the recoding; thus, the percent agreement was $280/306 = 91.5\%$. This metric is intuitive and easy to calculate, but it is also problematic. In particular, it does not account for agreement that would occur merely by chance. Thus, the metric is biased in favor of coding frames with a small number of categories, since by chance alone, there would be a higher rate of agreement on a two-category coding frame than on a forty-category coding frame.

To address this problem, researchers developed more sophisticated interrater agreement measures, which correct for both the number of categories in the coding frame and the frequency with which they are used. The most important of these are Scott’s π [198], Cohen’s κ [54], and Krippendorff’s α [120; 122]. All of these are calculated by $(P_o - P_e)/(1 - P_e)$, where P_o is the percent agreement as defined above and P_e is the percent agreement to be expected on the basis of chance. They differ in how the expected agreement P_e is calculated. Informally, these coefficients measure the degree to which interrater agreement exceeds the agreement that would be expected by chance. A value of 1 indicates complete agreement, a zero value indicates that the agreement is no better than chance, and a negative value indicates agreement worse than chance.

	% agreement	π	κ	α
Content analysis 1	91.5%	0.912	0.912	0.912
Content analysis 2: “Classification”	94.7%	0.936	0.936	0.936
Content analysis 2: “Presence in initial architecture”	91.8%	0.874	0.874	0.874
Content analysis 2: “Presence in target architecture”	93.8%	0.900	0.900	0.900

Table 8. *Intracoder reliability measures: percent agreement, Scott’s π, Cohen’s κ, Krippendorff’s α*

The differences among the use of these coefficients are subtle, and some methodologists have expressed strong opinions about their appropriateness or inappropriateness for various uses [121]. For our purposes, the distinctions are unimportant, since π, κ, and α are all approximately equal for our data.

Our intracoder reliability figures appear in table 8. These measures were calculated with ReCal2, an online utility for calculating reliability coefficients [90], and a subset of the measures was verified by hand in Microsoft Excel. For content analysis 1, only one set of measures is shown, since each coding unit could be assigned exactly one category from anywhere in the coding frame. In content analysis 1, there were 306 coding units and 45 categories (i.e., the total number of subcategories that appear in appendix B.1.2, excluding the residual categories “mot-?,” “cau-?,” “con-?,” “cha-?,” and “app-?,” which were never used). For content analysis 2, on the other hand, three sets of reliability measures are shown, one for each main category. This is because each coding unit in content analysis 2 could be assigned three categories—one subcategory of each main category. There were 208 coding units in content analysis 2. There were ten subcategories of the “Classification” category (including the residual category “c-?,” which *was* occasionally applied, unlike the residual categories in content analysis 1). There were three effective subcategories of the “Presence in initial architecture” category, and likewise for “Presence in target architecture”; in addition to the two that appear in appendix B.2.2 (“Present” and “Absent”), “Not coded” was treated as a third, implicit subcategory for the purpose of the reliability analysis. (Per the coding guide, coding units identified as corresponding to software elements such as components and connectors were to be coded as present or absent in the initial and target architectures, while coding units identified as corresponding to non–software elements such as constraints and containment relations were not to be so coded. However, it turns out that I failed to adhere to this protocol consistently. In particular, I often incorrectly treated containment relations as though they were software elements, and coded them for their presence or absence in the initial and target architectures. In fact, this particular coding error accounted for a large portion of the disagreements in content analysis 2, depressing the reliability figures in table 8.)

All twelve of the reliability coefficients in table 8 are between 0.87 and 0.94. But how are we to interpret these figures? Well, there are no universally accepted thresholds, but a number of methodologists have put forth guidelines. Krippendorff [122, p. 325] recommends $\alpha = 0.8$ as a reasonable cutoff, with figures as low as $\alpha = 0.667$ acceptable when drawing tentative conclusions and in cases where reliability is less important. Another well-known recommendation is that of Fleiss et al. [84, p. 604], who describe $\kappa > 0.75$ as indicating "excellent" agreement, $0.40 \le \kappa \le 0.75$ as indicating "fair to good" agreement, and $\kappa < 0.40$ as "poor." Various other rules of thumb have been proposed [107, § 6.2.1; 164, p. 143].

There are a couple of problems with directly applying any of these well-known and widely used guidelines here. First, these guidelines are intended for assessing interrater, not intrarater, reliability. It seems reasonable to hold intrarater reliability to a higher standard than interrater reliability, since one is more likely to agree with oneself (even one's past self) than with another person.

Second, these guidelines are intended primarily for use in quantitative research. Arguably different standards should be applied to qualitative content analysis. Schreier [195, p. 173] writes:

> In qualitative research, you will typically be dealing with meaning that requires a certain amount of interpretation. Also, coding frames in [qualitative content analysis] are often extensive, containing a large number of categories. In interpreting your coefficients of agreement, you should therefore never use guidelines from quantitative research as a cut-off criterion [...], as in: 'Oh, my kappa coefficient is below 0.40—it looks like I will have to scrap these categories'. Instead, what looks like a low coefficient should make you take another closer look both at your material and your coding frame. Perhaps, considering your material and the number of your categories, a comparatively low coefficient of agreement is acceptable—this is simply the best you can do.

Schreier may be incorrect in suggesting that a large number of categories justifies laxer standards for reliability coefficients, since chance-adjusted coefficients such as π, κ, and α already account for the number of categories. However, the degree of interpretation required to apply a coding frame is a very good reason to treat qualitative content analysis differently from quantitative content analysis. Even Neuendorf [164, p. 146], who is dismissive of qualitative content analysis generally, argues that content analyses that require significant interpretation should be subject to more relaxed reliability standards: "Objectivity is a much tougher criterion to achieve with latent than with manifest variables, and for this reason, we expect variables measuring latent content to receive generally lower reliability scores."

Although this discussion has not yielded any precise thresholds to anchor our interpretation of the reliability figures in table 8, it should by this point be clear that coefficients of 0.87 to 0.94 are fairly high by almost any standard. Even considering the fact that these are coefficients of intrarater, not interrater, agreement, it seems reasonable to conclude that we have adequately demonstrated stability.

This leaves the other main reliability criterion relevant to content analysis: reproducibility. Most commonly, reproducibility is measured through interrater agreement. In this sense, reproducibility is synonymous with terms such as "interrater reliability" and "intersubjectivity." Of course, with a single-coder study design, interrater reliability cannot be assessed. Schreier [195, p. 198] recommends that in a qualitative content analysis in which a single researcher is coding the data alone, a second round of coding (at least 10–14 days after the main coding) should be used as a substitute for using multiple coders to assess reliability, implying that intrarater agreement may be used as a substitute for interrater agreement in single-coder content analyses and that stability alone is a sufficient indicator of reliability in such cases. Ritsert [189, pp. 70–71] takes a similar position: "In an analysis by an individual, the important possibility of intersubjective agreement as to the coding process is precluded, but the methods of 'split half' or 'test retest' can still provide an individual with information on the consistency and reliability of his judgments."[9]

On the other hand, Krippendorff [122, p. 272] argues that stability "is too weak to serve as a reliability measure in content analysis. It cannot respond to individually stable idiosyncrasies, prejudices, ideological commitments, closed-mindedness, or consistent misinterpretations of given coding instructions and texts." On similar grounds, Neuendorf [164, p. 142] writes that "at least two coders need to participate in any human-coding content analysis."

It is thus fair to say that there is no consensus in the content analysis literature about the adequacy of stability as a measure of reliability, and it would be misleading to assert that a single-coder content analysis design with stability as the primary reliability measure is uncontentious. However, there are certainly methodologists who do consider it an acceptable choice, and indeed content analyses with single-coder designs are common in the literature.

However, even in the absence of intercoder agreement as a reliability measure, reproducibility and intersubjectivity remain important goals in principle. A content analysis should be reproducible at least in principle, even if no test of interrater agreement was carried out in practice. Fortunately, there are other ways of getting at this quality in the absence of multiple coders. A particularly helpful perspective comes from Steinke [209, § 3], who argues that qualitative research needs to be evaluated differently from quantitative research: "For qualitative research, unlike quantitative studies, the requirement of inter-subject *verifiability* cannot be applied. [...] What is appropriate in qualitative research is the requirement to produce an inter-subjective *comprehensibility* of the research process on the basis of which an evaluation of results can take place." Steinke goes on to suggest that this intersubjective comprehensibility can be demonstrated in three ways. First, and most importantly, the research process should be thoroughly documented so that "an external public is given the opportunity to follow the investigation step by step and to evaluate the research process." Second, interpretations can be cross-checked in groups. Steinke writes that a strong form of this method is peer debriefing, "where a project is discussed with colleagues who are not working on the same project." Third, the use of codified procedures can contribute greatly to intersubjectivity.

All three of these methods were used in abundance in this case study. If you have reached this point in this lengthy discussion, the thoroughness with which the

[9] Translation from German mine.

research process was documented is not in question. In addition, in the process of planning and carrying out this study, I consulted with many colleagues in my department who were uninvolved with the research. Finally, the content analysis was conducted in accordance with a rigorously constructed, comprehensively defined coding frame, which is reproduced in its entirety in appendix B. In Steinke's terms, the detailed description of the research procedure and the use and publication of codified procedures permits readers to evaluate the research process and hence the results on their own terms. The publication of the coding frame also serves a more direct replicability purpose: other researchers can adopt the coding frame and apply it to other data to assess the extent to which the coding frame itself is generalizable. (However, because the coding frame is heavily data-driven, it would likely need to be adapted to be effective for use with other data. For example, in this study, we identified "bridge," "broker," "bus," "queue," and "transfer" as indicators signifying connector elements, but a different data set might use different words, such as "channel," "link," "pipe," and "stream.")

Before concluding this discussion of reliability, it is useful to examine briefly how reliability is treated in typical content analyses in the literature. After all, our aim in applying content analysis in this case study is to adopt a methodology that is commonly used in other disciplines—not to advance the current state of practice in content analysis. Thus, irrespective of various methodologists' opinions about how reliability issues should ideally be handled and reported, it makes sense to see whether our treatment of reliability is in line with common practice.

Indeed, it does not take much investigation to see that the current state of practice with respect to the treatment of reliability in content analysis is unimpressive. There have been several surveys investigating the treatment of reliability in published content analyses. The most recent of which I am aware is Lombard et al.'s 2002 content analysis of content analyses [141], which found that only 69% of content analyses indexed in *Communication Abstracts* between 1994 and 1998 included any discussion of intercoder reliability. In those that did, only four sentences, on average, were devoted to discussion and reporting of reliability. Only 41% of the studies specifically reported reliability information for one or more variables, 6% had tables with reliability information, and 2% described how the reliability was computed. These results were broadly in line with earlier surveys. The state of practice may have improved somewhat since Lombard et al.'s survey, but it is still easy to find content analyses with no discussion of reliability.

Incidentally, since Lombard et al.'s survey was itself a content analysis, Lombard et al. reported their own reliability figures: percentage agreement, Scott's π, Cohen's κ, and Krippendorff's α. It later turned out that due to bugs in the software they had used to calculate these figures, they had themselves reported erroneous reliability values [142]—an amusing illustration of the difficulty of getting reliability reporting right. (I verified many of my reliability figures by hand to avoid any similar embarrassment.)

In light of such results, it seems that our detailed treatment of reliability here puts us ahead of the current state of practice in typical content analyses, even in the absence of any measurement of intercoder reliability.

5.6.3 Validity

The situation with validity is more muddled than that with reliability. There are just three main types of reliability that are relevant to content analysis—stability, reproducibility, and accuracy—and reliability can be easily quantified via well-accepted measures of coding agreement. But there is a bewildering array of flavors of validity: internal validity, external validity, construct validity, content validity, face validity, social validity, criterion validity, instrumental validity, sampling validity, semantic validity, structural validity, functional validity, correlative validity, concurrent validity, convergent validity, discriminant validity, predictive validity, ecological validity. All of these, and others, have been described as relevant to content analysis. To evaluate the validity of our content analysis, it is necessary to untangle this knot of concepts and focus on the elements of validity most relevant to this research.

Krippendorff [122, ch. 13] identifies three broad categories of validity at the highest level: face validity, social validity, and empirical validity.

Face validity. Krippendorff [122, p. 330] describes face validity as "the gatekeeper for all other kinds of validity," so it is a good place to start. Face validity is the extent to which an instrument appears on its face to measure what it purports to measure. Face validity thus appeals to a commonsensical evaluation of the plausibility of a study. Neuendorf [164, p. 115] explains:

> It's instructive to take a "WYSIWYG" (what you see is what you get) approach to face validity. If we say we're measuring verbal aggression, then we expect to see measures of yelling, insulting, harassing, and the like. We do not expect to find measures of lying; although a negative verbal behavior, it doesn't seem to fit the "aggression" portion of the concept.

In the context of a content analysis such as this one, face validity may be assessed by considering whether the coding procedure appears to correspond with the concept being measured—the research questions. In section 5.6.1, I examined how our research questions are addressed by the findings of the content analysis (and the modeling phase that followed the content analysis). In addition, the coding guide appears in appendix B. By direct examination of the coding guide, it can be seen that the coding frame appears on its face to capture the concepts it sets out to capture.

Face validity, however, is a low bar—the mere *appearance* of measuring the correct concept. Weber [220, p. 19] calls it "perhaps the weakest form of validity." I now continue to examine other kinds of validity relevant to content analysis.

Social validity. The term "social validity" was introduced in 1978 by Wolf [224] in the field of applied behavior analysis. In its original sense, the term refers to how well a social intervention is accepted by those who are meant to benefit from it. This sense is relevant only to interventionist methodologies such as action research.

In content analysis, social validity has a rather different meaning. (For a discussion of how a variety of meanings of "social validity" arose subsequent to the initial introduction of the term, see Schwartz & Baer [196].) A useful definition is given by Riffe et al. [188, p. 137], who describe social validity as hinging on "the degree to which the content analysis categories created by researchers have relevance and meaning beyond an academic audience.” Social validity thus gets at the *importance* of research.

While social validity is a criterion that has seldom (if ever) been applied to software engineering research, the idea of ensuring that research is relevant to the real world is an important one. I have motivated our work here in terms of its relevance to practicing software architects. Software architects today, I have argued, face substantial challenges in planning and executing major evolutions of software systems, and our work aims to help address a subset of these challenges. Our work thus has social validity to the extent it succeeds in having relevance to software practitioners.

Empirical validity. Krippendorff uses “empirical validity” as an umbrella term to encompass a number of specific types of validity. In general, empirical validity gets at how well the various inferences made within a research process are supported by available evidence and established theory, and how well the results can withstand challenges based on additional data or new observations. Assessments of empirical validity are based on rational scientific considerations, as opposed to appearances (face validity) or social implications (social validity). Empirical validity is a broad concept with a number of distinct facets, which I will now proceed to examine.

There are several conventionally recognized types of validity that fall under the heading of empirical validity: content validity, construct validity, and criterion validity. Although some methodologists have undertaken critical examinations of challenges in applying these (most notably, Krippendorff [122, ch. 13], after introducing and explaining these concepts, introduces his own typology of validation for quantitative content analysis), this trio of validities remains very popular in the literature and continues to be embraced by many content analysis methodologists [164, pp. 115–118; 195, p. 185].

Content validity. Content validity is the extent to which an instrument captures all the features of the concept it is intended to measure. As Schreier [195, pp. 186–191] notes, in a qualitative content analysis, content validity is of relevance to *concept-driven* (deductive) coding frames. Data-driven (inductive) coding frames are instead concerned with face validity. (The reason for this should be obvious. When the categories are generated from the data, instead of being derived from a theory, there is no expectation that they should capture any particular theoretical concept; instead, the only concern is that they faithfully describe the data. But when categories are derived from a theoretical concept, content validity is a concern, because it needs to be shown that the categories adequately cover the concepts they purport to cover.) In this study, only content analysis 2 used a deductive coding frame; the coding frame for content analysis 1 was data-driven.

Here, the issue of content validity is a fairly straightforward one, because the categories for content analysis 2 were taken directly from the key concepts in our research—evolution operators, evolution constraints, and evolution path analyses (called dimensions of concern in the coding frame)—as well as traditional software architecture concepts such as components and connectors. (I consider here only the

This category should be applied to software elements that are best characterized as *connectors*.	—
A connector represents a pathway of interaction between two or more components.	A **connector** is a runtime pathway of interaction between two or more components. [53, p. 128]
Examples of kinds of connectors include pipes, streams, buses, message queues, and other kinds of communication channels.	Simple examples of connectors are service invocation, asynchronous message queues, event multicast, and pipes that represent asynchronous, order-preserving data streams. [53, p. 128]
Although connectors are often thought of as simple, some connectors are quite complex, and connectors can even have an internal substructure, just as components can. Thus, do not categorize an element as a component simply because it seems complex and bulky.	Like components, complex connectors may in turn be decomposed into collections of components and connectors that describe the architectural substructure of those connectors. [53, p. 128]
Instead, the determination of whether an element is a component or a connector should be guided by its function. If it is principally a computational or data-processing element, it should be categorized as a component. If its principal role is to facilitate communication between components, it should be categorized as a connector.	If a component's primary purpose is to mediate interaction between a set of components, consider representing it as a connector. Such components are often best modeled as part of the communication infrastructure. [53, p. 129]

Table 9. *The description for the "Connector" category in content analysis 2 (left column), juxtaposed with selected excerpts from* Documenting Software Architectures *[53] (right column). Basing our category definitions on widely accepted descriptions of the concepts under study bolsters their content validity.*

"Classification" main category of content analysis 2; the other two main categories are trivial and straightforward.)

For this study, then, the issue of content validity hinges on whether the category definitions in appendix B.2.2 adequately capture the concepts that they claim to capture. The descriptions of general software architecture concepts such as components, connectors, and attachment relations in the coding guide were based heavily on a widely used text on architectural representation, *Documenting Software Architectures* [53]. For example, compare the description of the "Connector" category in content analysis 2 with selected passages from the text of *Documenting Software Architectures* in table 9. Descriptions of the categories capturing concepts from our approach (operators, constraints, analyses) were similarly based on our previous own descriptions of those concepts.

Construct validity. Construct validity is “the extent to which a particular measure relates to other measures consistent with theoretically derived hypotheses concerning the concepts (or constructs) that are being measured” [43, p. 23]. As Krippendorff [122, p. 331] explains, this concept “acknowledges that many concepts in the social sciences—such as self-esteem, alienation, and ethnic prejudice—are abstract and cannot be observed directly.” In this study, we are measuring the concepts we wish to measure rather directly; we are not using easily observable qualities as a proxy for difficult-to-observe qualities. Thus, construct validity is of little relevance here.

Criterion validity. A procedure has criterion validity (also called instrumental validity or pragmatic validity) “if it can be shown that observations match those generated by an alternative procedure that is itself accepted as valid” [117, p. 22]. Like construct validity, criterion validity is of limited relevance here. There are no accepted procedures for measuring the qualities that our coding frame seeks to get at, and hence no basis for assessing or discussing criterion validity.

There is one final type of validity that is highly relevant here: external validity,[10] or generalizability. Determining the appropriate scope of inference—the degree to which results can be generalized—is one of the most challenging and important aspects of the validation of a content analysis that seeks to have meaning beyond the immediate context of the data it is based on, or of a case study that aims to have implications beyond the case it examines. Since this work is both a content analysis and a case study (or more precisely a case study that incorporates a content analysis), the problem of generalizability is particularly acute.

It is helpful to bear in mind a few points. First, generalizability is not all-or-nothing. A study is not “generalizable” or “ungeneralizable.” Rather, a study is generalizable to a certain extent—in certain respects and to certain contexts. The question is not whether this case study is generalizable, but to what extent (in what respects, to which contexts) it is generalizable.

Second, a case study is not generalizable in the same sense that a study based on statistical sampling is generalizable. In a typical observational study or controlled experiment, generalization is straightforward. The study is conducted on a sample of some population, and the study is generalizable to the extent that the sample is representative of the population. In a case study, the goal is not statistical generalizability, but instead some notion of *transferability* or *analytic generalizability*.[11] The

[10]The appearance of “external validity” here may lead you to wonder why the corresponding term “internal validity” appears nowhere in this section. Internal validity is the extent to which a study adequately demonstrates the causal relationships it purports to demonstrate. Since we are not making any causal claims—this is a descriptive and evaluative case study, not an explanatory one—there is no issue of internal validity.

[11]There is considerable disagreement about the proper handling of generalization in case studies and qualitative research generally. A summary of different schools of thought is given by Seale [199, ch. 8]. There are two particularly significant such schools, which I will summarize in this footnote.

The first camp holds that the proper criterion for generalizability of qualitative research is *transferability*. The goal of a case study, these methodologists say, is not to discover some kind of law that is of general applicability to the world. Rather, it is to achieve a rich understanding of a single case studied is unique, but the findings of the case study can still be applied to other contexts.

Finally, there is another sense in which all-or-nothing thinking is unhelpful in reasoning about generalizability. Not only are there shades of gray in the generalizability of the case study as a whole, but also different results of a single case study may be generalizable in different ways. Some results may be highly specific to the context of the case study, while other may be readily transferable to other cases.

With these points in mind, we can examine the issue of generalizability with respect to our results. In evaluating analytic generalizability or transferability, we must consider what special characteristics of the case may have influenced the results. For example, in this case study, "Dealing with legacy systems" was among the most significant challenges of architecture evolution mentioned by participants. However, these legacy challenges have a great deal to do with the history of the specific company under study. At companies whose software systems have a similar history—companies with large, complex software systems that date back decades, built on mainframes using now-archaic technologies—this result would be transferable. And in fact, there are many companies with such a history. But at a company whose software systems have a very different history, the result would likely be quite different. Consider Amazon.com, for example. Amazon.com certainly has challenges with legacy systems, but they are quite different in character to those that Costco faces. Amazon.com is a significantly younger company, and even it's oldest systems are at worst dated, not obsolete. And of course, a new start-up would have no legacy challenges at all.

On the other hand, the number one architecture evolution challenge that emerged in this study was "Communication, coordination, and integration challenges." There is no clear a priori reason to expect particularly acute communication, coordination, and integration challenges to arise in this case. On the contrary, the organization studied has taken significant measures in recent years to ameliorate communication,

case through thick, naturalistic description, such that the results of the case study can be transferred on a case-by-case basis to other contexts where they may be applicable. If we accept this notion of transferability, then transferring the results of a case study requires not only an understanding of the context of the case study itself, but also an understanding of the context to which the results are to be transferred. Significant methodologists in this camp include Lincoln & Guba [139] and LeCompte & Goetz [134].

In the second camp are methodologists who espouse some notion of theoretical generalization. Most prominent among these is Yin [226, pp. 38–39], who argues that a case study can serve as a way of validating a previously developed theory, an approach which he calls *analytic generalization.* Theoretical notions of case study generalizability are stronger than transferability because they generate or provide support for theories that have broad applicability well beyond the immediate context of the case study. However, some methodologists have criticized theoretical notions of generalizability due to the difficulty of deriving broad theoretical results from individual cases. Seale [199, p. 112], for example, is critical of theoretical conceptions of generalization, arguing that transferability "is a more secure basis for good work."

Of course, these viewpoints are not mutually exclusive. Some methodologists advocate consideration of both transferability and analytic generalizability [125, pp. 260–265; 158, p. 279]. In my discussion of the generalizability of this case study, I have avoided taking a stand on the proper understanding of generalization in qualitative research. But it is useful to keep these schools of thought in mind when considering how this work might be generalizable.

coordination, and integration difficulties, but still there are significant challenges. Indeed, on a theoretical basis, we might expect that challenges in integrating systems and communicating with relevant personnel are highly relevant to architecture evolutions in general. Thus, we might reasonably say that our result on the prominence of communication, coordination, and integration challenges is more analytically generalizable than that on dealing with legacy systems.

Of particular interest is the generalizability of our results on the applicability of our approach to modeling a real-world evolution. Generalizability is a particularly crucial question here, because not only does this result emerge from a study at a single company, but it emerges from a study of just a single software system. To what extent is this result generalizable?

Again, it is important to keep in mind how generalization works for case studies. There are some aspects of the case under study that do limit generalization in certain respects. For example, the point-of-sale system is one that can be understood fairly well from a component-and-connector architectural perspective. Our coding frame and modeling approach took advantage of this fact. The categories of the coding frame are based on the assumption of a component-and-connector model, and we produced only a component-and-connector view during the modeling process. In a different system where alternate architectural view types are important, the coding frame and the modeling procedures would have to be revised accordingly.

But the overall result—that our approach can capture the main elements of a real-world evolution—seems to have a good deal of generalizability. There was no a priori theoretical reason to believe that the point-of-sale evolution would be particularly easy to model using our approach. We picked that evolution because it was of a reasonable size for the purposes of the case study, and because it was easy to get access to key personnel involved with architecting the system, not because of any special properties that would render it more amenable to our approach. Thus, the main evaluative result seems to have fairly strong external validity. But of course this generalizability has limits. For example, it would be questionable to transfer this result to evolutions of a very different scale (extremely large and complex evolutions, or small code-level evolutions), or to evolutions with special properties that we have theoretical reasons to believe might be difficult to model (e.g., significant elements of uncertainty). This is not to say that such evolutions could not be modeled using our approach—only that this case study does not clearly demonstrate that they can be. Ultimately, case study generalization involves a clear understanding of relevant theory, careful attention to the specifics of both the case being generalized and the case to which the result is to be transferred, and a good deal of judgment.

5.7 Summary

In section 5.6, I explained how the research questions articulated at the beginning of this chapter were answered by the case study. But how do these case study research questions and results relate to the broader purpose of the thesis—the thesis research questions listed in section 1.1 and the thesis claims listed in section 1.3.3? Let us again turn to each of the case study research questions and consider its relevance to the thesis.

1. **How do practicing architects in a real-world software organization plan and reason about evolution?** With respect to the thesis, answering this question was important for ensuring that architects do not already have approaches and tools similar to ours for managing architecture evolution—that we haven't reinvented something that already exists in industry, or addressed a need that has already been adequately met. This question thus isn't directly germane to any of the thesis claims in section 1.3.3, but rather helps to ensure that the foundation for this research is secure—that the thesis research questions asked in section 1.1 haven't already been answered by industrial approaches.

   In this respect, the answer to this research question was generally encouraging. Although architects mentioned a variety of approaches and tools that they employ in the course of planning and carrying out evolution projects, they did not report having tools to help them with things like architectural planning of evolution stages, enforcement of constraints, and evaluation of architectural trade-offs. Most of the approaches that architects mentioned were general strategies for communication, coordination, and organization rather than concrete approaches or tools; and to the extent that they did describe concrete approaches and tools, those approaches and tools had very different purposes than the purposes that motivate this thesis work.

2. **What difficulties do practicing architects face in planning and carrying out evolution?** Like the previous case study research question, this question does not directly address the thesis claims, but instead helps to evaluate whether this research is well-founded—whether it addresses a genuine need perceived by practicing architects. Where case study research question 1 can help us to ensure that the thesis questions in section 1.1 haven't already been answered by industrial approaches, case study research question 2 can help us to ensure that the thesis questions are worth answering in the first place.

   The case study revealed a wide range of challenges pertaining to architecture evolution. Many of these—integration challenges, justifying architectural decisions in business terms, evaluating reliability and performance, domain inexperience, and others—are challenges that our approach is intended to ameliorate. Others, such as managing people and cultural challenges, are of little relevance to us. But on the whole, the results of the case study are consistent with the premise that the challenges that motivate our research are genuine challenges faced by practicing architects in (at least some) real software organizations.

3. **How well can our modeling framework capture the concerns that arise in a real-world architecture evolution?** This case study research question is the one that is of clearest relevance to the thesis claims in section 1.3.3. Specifically, this research question directly addresses the claim of applicability—the claim that our approach is applicable to the concerns that arise in real-world

architecture evolution projects. The fact that we were able to easily and directly model nearly all of the architectural concerns in the point-of-sale evolution provides strong support for the applicability claim.

The preceding discussion clarifies how the case study described in this chapter relates to the thesis as a whole. But how does this case study relate to the JPL case study in chapter 4? After all, both case studies evaluate the applicability of our modeling approach to real-world architecture evolution scenarios. That is, both case studies directly address thesis claim 1 in section 1.3.3: the claim of applicability. Why did we carry out two case studies addressing the same topic?

One answer is that two case studies are much stronger than one. Critics can dismiss the result of a single case study as a fluke attributable to the unique characteristics of the case studied rather than evidence of any broader truth. When two different case studies converge on similar findings, the results are harder to dispute. This kind of replication logic or triangulation can be quite powerful, especially when the two case studies were carried out in particularly dissimilar organizations. The two cases studied here are very different indeed: an aerospace laboratory and a retailer. These two organizations have very different goals (exploring space vs. making a profit), very different types of systems (remote cyberphysical systems vs. enterprise software systems), very different notions of system criticality (supporting billion-dollar missions and sometimes sustaining human life vs. avoiding lost sales), and very different challenges of scale (millions of miles vs. millions of users), among many other differences. Examining two quite different cases, rather than only a single case, greatly strengthens the claim of applicability.

Beyond this, however, it is worthwhile to examine the differences between the two case studies we carried out, because in fact they are really quite different. Although they both examine the applicability of our approach and in particular the question of whether we can successfully represent real-world evolution concerns, they have very different designs and focuses. Table 10 summarizes the differences.

One important difference is that the two case studies had somewhat different goals: one focused on examining how our approach could be adapted to industrial architecture modeling tools, while the other was concerned with examining the practice of architecture evolution and the applicability of the elements of our approach. Both aimed to evaluate the applicability of our modeling approach to a real-world architecture evolution. But a secondary goal of the first case study was to evaluate the ease of adapting our approach to modeling languages and tools already in use at JPL—namely SysML and UML. By contrast, this was not a goal of the second case study at all; modeling in the second case study was done in AcmeStudio, a research tool. The second case study had a different secondary goal: to examine in depth how a real-world software organization currently manages architecture evolution challenges. The Costco case study thus had a significant descriptive or ethnographic component, which was not true of the JPL case study.

The data collection procedures were also quite different. In the first case study, data was collected informally during the course of a ten-week internship at JPL. In the second, there was a formal data collection procedure involving semistructured

Chapter	Chapter 4	Chapter 5
Organization studied	Federally funded laboratory	Major retailer
Research questions	Can an evolution graph capture the major concerns and alternatives under consideration in a real-world evolution? Can we adapt our approach for use with commercial modeling languages and tools?	Can we use operators, path constraints, and path evaluation functions to capture real-world evolution concerns? What major challenges do architects face in planning and carrying out evolution, and how do they deal with them today?
Data collection approach	Information and impressions gathered informally during a ten-week internship	Formal semistructured research interviews during a two-week visit
Analysis approach	Ad hoc construction of an evolution graph	Formal content analysis leading to the construction of a partial evolution model including operators, constraints, and evaluation functions
Publication	[16]	[17]

Table 10. *A summary of the two case studies that together constitute the empirical validation of my thesis, emphasizing the differences among them.*

interviews of architects supplemented by collection of architectural documentation.

The two case studies also used very different analysis procedures. The first case study lacked any formal analysis procedure; I constructed an evolution model ad hoc based on my impressions of the system under study. The second case study used a fairly sophisticated analytical approach involving a formal content analysis of the gathered research data followed by a modeling phase incorporating the results of the content analysis. The application of content analysis for extracting architectural information is novel, and I regard this methodological technique as one of the ancillary contributions of this thesis (see section 8.1).

Finally, the second case study devoted significantly more attention to methodological issues in general. The first case study was conducted opportunistically during a summer internship and was not strictly guided by a formal design (although there was some consideration of qualities such as research replicability). This first case study was exploratory in character; the focus was on discovering new ways of adapting our approach to a real-world setting, unencumbered by the more heavyweight methodology that characterized the second case study. In the second case study, great care was taken to develop a detailed, rigorous case study design to guide the execution of the case study, and significant attention was devoted to issues of validity and reliability. A key goal of the study was to ensure that the conclusions of that case study were reliably tied to the data collected. Thus, although the two case studies share the same basic motivation, they distinguish themselves clearly in their design and execution; the first case study was more formative in character, while the second was more summative.

6 Tooling

Chapter 3 presented a theoretical analysis of our evolution path constraint specification language, thereby providing an assessment of the *computational complexity* of our approach. Chapters 4 and 5 described our empirical work, aimed at evaluating the approach's *applicability*. In this chapter, I address the last remaining research claim from section 1.3.3: the claim of *implementability*.

I do so by reviewing three prototype tools that we have developed based on the approach to software architecture evolution described in this thesis. None of these prototype tools constitutes a complete evaluation of the implementability of our approach by itself; each of the three tool development projects was intended to implement only a limited subset of the approach. But together these three projects, which are very different in their focus and technical approach, reveal a great deal about the issues that arise in implementing an evolution modeling tool based on our approach.

The first of the three prototypes, called Ævol, was developed as a plug-in to AcmeStudio. This was very early exploratory tool work with which I was only very distantly involved, so I will not say much about it except to provide some relevant background. I describe this work briefly in section 6.1.

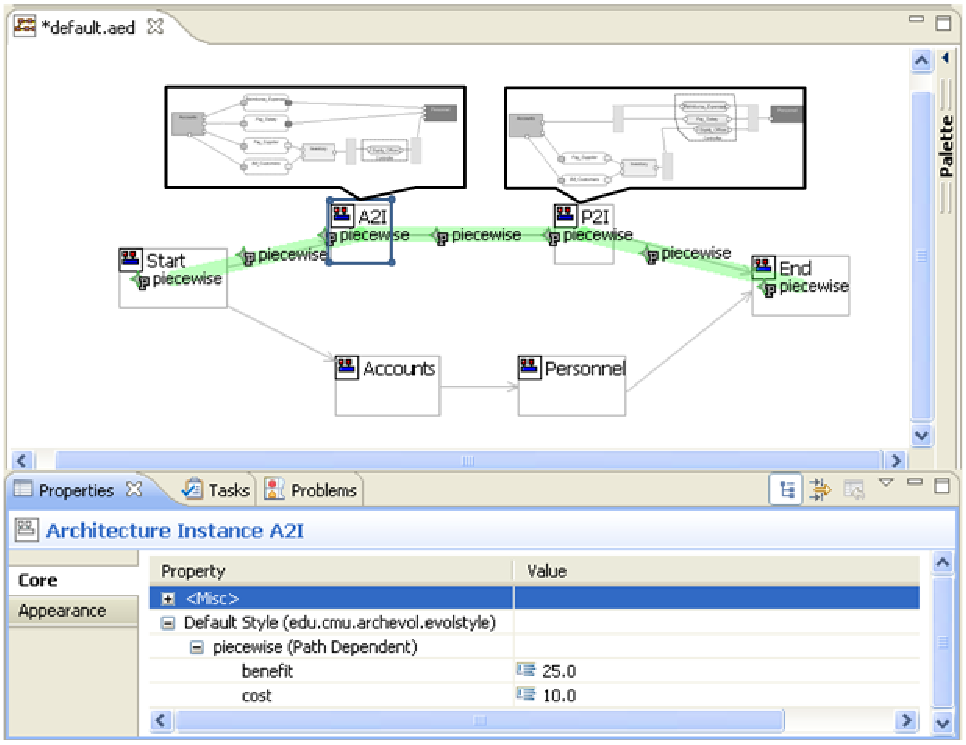
The second prototype was an extension of the JPL case study described in chapter 4. In this second prototype project, we developed a plug-in to MagicDraw (the commercial UML tool used in the JPL work) implementing a subset of our approach. One of the main goals of this work was to examine the engineering challenges involved in implementing an architecture evolution tool as a plug-in to an existing architecture modeling framework. This work is described in section 6.2.

The final prototype was quite different in character from the first two. It was narrowly focused on evaluating the *automatability* of evolution path generation. This work aimed to demonstrate how existing tools based on automated-planning research could be applied to generate evolution paths leading from a given initial architectural state to a given target architectural state by intelligently composing evolution operators from a predefined library. I describe this work in section 6.3.

In section 6.4, I will summarize this chapter by comparing these three pieces of work. Although all of them address research questions about implementing architecture evolution tools, they have very different focuses and very different perspectives.

6.1 Ævol: A first step in tool development

The first attempt to implement a prototype based on our approach was an AcmeStudio plug-in called Ævol, which was developed by a team of master's students as



Property	Value
<Misc>	
Default Style (edu.cmu.archevol.evolstyle)	
piecewise (Path Dependent)	
benefit	25.0
cost	10.0

Figure 14. *The Ævol workbench.*

a studio project. I was only peripherally involved with the work on the Ævol tool, and therefore it does not constitute a contribution of my thesis. Nonetheless, it is useful to briefly review this work, because it provides useful information about the implementability of our approach in its own right, and discussing this early work also provides helpful background for our subsequent tooling work to be discussed in sections 6.2 and 6.3. However, because of my limited involvement with this work, I discuss it only briefly here. For a detailed description of the Ævol tool, see Garlan & Schmerl [96].[12]

Because Ævol is a plug-in to AcmeStudio, it is able to take advantage of AcmeStudio's existing architecture modeling facilities. Thus, individual evolution states are modeled as architectures using AcmeStudio, and the Ævol plug-in relates them by means of an evolution graph. A screenshot of the Ævol workbench appears in figure 14. Ævol does not provide any support for evolution operators. This means there is no way of defining evolution states by applying transformations to other states; each node in the evolution graph has to be defined in isolation, either by building it from scratch or by copying, pasting, and modifying another node.

[12]The development of the Ævol tool was carried out by Snehal Fulzele, Smita Ramteke, Ken Tamagawa, and Sahawut Wesaractchakit as a studio project in our Master's in Software Engineering program, under the direction of David Garlan and Bradley Schmerl [96]. A second studio project to develop additional functionality was later carried out by another team of master's students: Andrew O. Mellinger, Mohit Bhonde, Raúl A. Véjar Llugany, Majid Alfifi, and Adlan Israilov.

However, once the nodes of the evolution graph are defined, Ævol provides support for definition of evolution paths as well as some basic support for definition of constraints and evaluation functions. Figure 14 shows a rudimentary evaluation function that estimates the overall cost and benefit of a path by adding up cost and benefit properties defined on the nodes and transitions of the evolution graph.

This early tool work provided a helpful illustration showing how our approach could be implemented in a tool, but it was ultimately a proof of concept rather than a careful evaluation of the implementability of the approach. In the rest of this chapter, I will describe two implementation efforts with which I was more closely involved.

6.2 A MagicDraw plug-in for architecture evolution

Our second implementation effort was conceived as a follow-on project to the JPL case study that I described in chapter 4. Recall that in that work I demonstrated how our approach could be adapted to MagicDraw, a commercial UML tool that is widely used at JPL. To do so, I developed ways of modeling architecture evolution in MagicDraw, including representational conventions for modeling architecture evolutions in UML (or, more precisely, SysML), model transformation macros to implement evolution operators, and SysML models of particular intermediate states of evolution. Together, these mechanisms served as an illustration of how our approach could be adapted to commercially prevalent languages and tools such as UML and MagicDraw.

That work, like the early tool development work I described in section 6.1, was a proof of concept. It demonstrated how our approach could be implemented in principle, but it was not conceived as an in-depth investigation of the *implementability* of our approach. Rather, the JPL case study was intended to evaluate the *applicability* of our approach to a real-world evolution context; implementation issues were a secondary consideration. Similarly, the Ævol tool development work, although it did seek to show that our approach was implementable, did not involve significant reflection on the practical issues involved in implementation. The prototype work I describe in this section, on the other hand, was a more deliberate examination of the challenges involved in developing a practical architecture evolution tool.[13]

To extend MagicDraw so that it could be used as a general tool for evolution planning, we recognized that it would be necessary to take a plug-in-based approach, much like the Ævol prototype described in section 6.1. A plug-in-based approach has several advantages. First, developing an architecture evolution tool as a plug-in to a tool such as MagicDraw allows us to leverage the facilities that tool provides for architectural modeling. This frees us to focus our development efforts on implementing the novel aspects of our approach, rather than reinvent the wheel by building our

[13]The development of the plug-in described in this section was carried out in 2012 by Nivedhitha Manjappur Narayanaswamy and Vignesh Venkatachala Perumal as a studio project in our Master's in Software Engineering program, supervised by my advisor and me. In 2013, we published a reflection on this effort as a workshop paper [18]. Some of the material in this section is adapted from that workshop paper.

own framework for architecture modeling. In addition, a plug-in-based approach can facilitate user adoption, because users are already likely familiar with the interface and idioms of the framework on which the plug-in is based (particularly a widely used framework such as MagicDraw). Finally, plug-ins are significantly more powerful than other extension mechanisms, such as the macro-based approach used in the JPL case study, allowing us much more freedom in implementing our approach.

Perhaps the best way to describe the main features we envisioned in a plug-in for evolution planning is to consider a sample work flow, illustrating how a software architect might use such a plug-in to define and analyze an evolution space:

1. The architect begins by defining the nodes and transitions of the evolution graph. This can be done using the architectural modeling tools already present in MagicDraw. As in the JPL case study described in chapter 4, we can represent an evolution graph as a package diagram in which evolution states are represented by packages and transitions are modeled as dependencies. To define the evolution graph, the architect simply creates a package for each intermediate state and draws the transitions as dependencies among the packages. Once the graph is represented in this way, the plug-in can use the MagicDraw API to inspect the evolution graph, then determine the set of candidate evolution paths.

2. At this point, the evolution graph has been laid out, but all the evolution nodes are undefined, in the sense that all the packages representing intermediate states are empty—they do not yet contain representations of intermediate architectures. The architect must now “fill in” the evolution graph by specifying these intermediate nodes. The first node that the architect fills in is the initial node, which represents the system architecture at the outset of the evolution. (A plug-in can identify the initial node algorithmically. The initial node is simply the graph’s source node—the node with in-degree 0, which should be unique in a well-formed evolution graph.) To do so, the architect uses MagicDraw’s existing facilities to create an architectural model contained within the package representing the initial node.

3. Now that the first node has been “filled in”—the initial architecture has been specified—the architect can begin using the plug-in to define the other intermediate states. To do so, the architect selects one of the transitions leaving the initial state, then clicks a “Specify Transition” button provided by the plug-in. This launches transition specification mode, in which the architect applies a sequence of operators capturing the architectural transformations that make up the transition. The architect selects these operators from a predefined palette and specifies the parameters necessary to apply them to the evolving model. As the architect does so, the effects of the operators are displayed. Once the definition of the operators making up the transition is finished, the architect exits transition specification mode. The newly defined architecture is now associated with the relevant state in the evolution graph; thus, there are now two nodes “filled in.”

4. The architect continues specifying transitions as in step 3 until all transitions in the graph have been specified. At this point, all nodes have been "filled in."
5. With the evolution graph fully defined, the architect now clicks a "Validate Evolution Paths" button, which causes the plug-in to validate all the evolution paths in the graph with respect to a predefined set of constraints. If a path fails to satisfy any of the constraints, the user is notified of the violation and given the opportunity to correct it.
6. Finally, the architect can run a set of predefined evaluation functions, which provide quantitative assessments of each valid path. The architect is now equipped to make an informed decision about the optimal evolution path.

With this operational vision in mind, we set out to create a new prototype, the development of which would allow us to explore the significant implementation issues involved in developing an architecture evolution planning tool in a much deeper way than had been done in the development of the Ævol tool or in the JPL case study.

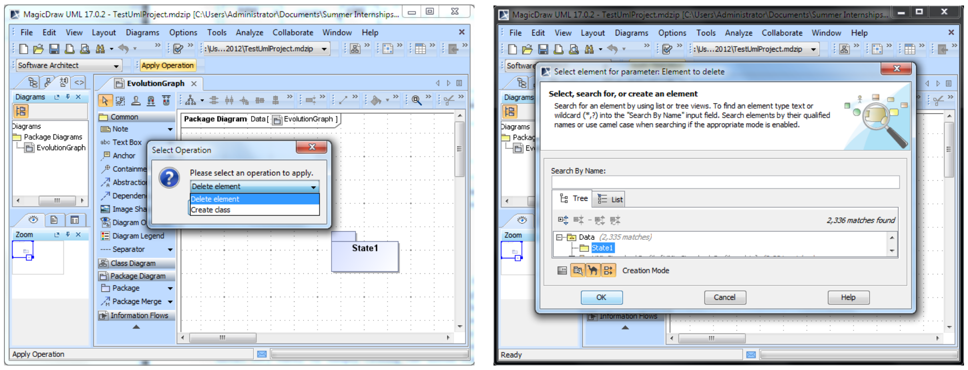
6.2.1 Developing the plug-in

There are several ways in which the functionality of MagicDraw can be extended. The most basic and limited is by defining *UML profiles*. This allows users to enrich the available modeling vocabulary by extending UML itself; however, UML profiles cannot be used to modify the user interface or behavior of MagicDraw. A much richer extension mechanism is the *MagicDraw Macro Engine*, which I used in our JPL case study to generate an evolution graph, as described in chapter 4. Finally, the most powerful means of extension that MagicDraw provides are *plug-ins*, the approach we used to develop this prototype.

Both macros and plug-ins interact with MagicDraw via the *MagicDraw OpenAPI* [166], a Java API that provides numerous means of extending or enhancing MagicDraw's functionality. With the OpenAPI, it is straightforward to add user interface features (such as buttons and menu items), manipulate model and presentation elements, listen for events, and so on.

Development of the prototype proceeded in stages, each designed to result in a particular unit of functionality. These units—"feature prototypes," we called them—would be pieced together at the end of the project, in a final integration phase. This staged development approach allowed us to familiarize ourselves with the relevant portions of the OpenAPI rapidly. In addition, developing the plug-in by way of a series of partial prototypes made sense because the plug-in's responsibilities were clearly delineated; thus, the components of the plug-in's planned architecture were well defined and interacted with each other in well-understood ways. The development stages were:

Feature 1: Manipulation of the evolution model. Our first feature prototype was intended to familiarize us with the fundamentals of MagicDraw plug-in development and to acquaint us with MagicDraw's model manipulation API. We therefore created


Select Operation

Please select an operation to apply.

Delete element

Select element for parameter: Element to delete

Select, search for, or create an element

Search for an element by using list or tree views. To find an element type text or wildcard (*,?) into the "Search By Name" input field. Search elements by their qualified names or use camel case when searching if the appropriate mode is enabled.

Search By Name:

2,336 matches found

Data (2,335 matches)

State1

Figure 15. *Screenshots of feature prototype 3, the operator palette interface. At left, the user selects an operator from the palette. In this demo, two operators are defined. The user selects "Delete element." At right, the user is prompted to specify the parameter required by the operator: the element to be deleted.*

a simple plug-in that made a predefined sequence of API calls to manipulate a predefined evolution model in MagicDraw. This prototype was similar in function to the transformation macro that was used to define evolution operators in the JPL case study. Here, however, this was accomplished by means of a plug-in rather than a macro.

Feature 2: Identification of evolution paths. The second feature prototype was intended to demonstrate basic reasoning about an evolution model. Given a MagicDraw model of an evolution graph, this feature prototype would analyze the model and identify all the evolution paths in the graph.

Feature 3: Interface for applying evolution operators. Next, we wanted to familiarize ourselves with MagicDraw's user interface extension capabilities. The third feature prototype was thus an interface for applying an evolution operator (Fig. 15). To apply an operator, the user firsts select an operator from a palette, then specifies any required parameters. Each operator can define an arbitrary set of typed parameters. For example, a "delete element" operator might take one parameter: the element to be deleted. A "create component" operator might have multiple parameters: the name of the component, its type, and so on.

Feature 4: Operator parser. We wanted operators to be specifiable via a configuration file that would be parsed and interpreted at the time the plug-in is loaded, so that plug-in users could add, remove, or modify operators easily. In this stage, we developed a parser for a highly simplified version of our operator specification language.

Feature 5: Metadata handling. This feature prototype focused on storage of *metadata* pertaining to the model—that is, information that is auxiliary to the evolution model but necessary to support analysis, such as the *sequence of evolution operators* that makes up each transition in the model. We considered several mechanisms for storing metadata. The most principled approach would have been to develop a UML profile for the purpose of representing architecture evolution graphs. However, while elegant, this solution would have taken too long to implement given the limited time available. For this feature, we adopted the cruder but more expedient approach of storing metadata in existing fields intended for other purposes.

Feature 6: Positioning of presentation elements. A significant challenge of developing an evolution planning tool as a plug-in to a tool like MagicDraw is dealing with the positioning of presentational elements in intermediate states. For example, when a user applies an operator that introduces a new component, how should we position that component in any diagrams that are associated with the new state? We considered a number of possible approaches, including making use of the layout algorithms built into MagicDraw, requiring the user to specify the placement of new presentation elements, or even developing layout algorithms of our own. Ultimately, we again adopted the most expeditious approach; to determine the layout of an evolution state, we cloned the layout of the state from which it was derived, then allowed MagicDraw to position any new elements in the default location.

Feature 7: Evolution path constraints. Evolution path constraints, like operators, are defined in a configuration file that is read by our plug-in when it is loaded. The constraints are written in a simplified subset of our constraint specification language. The final feature prototype comprised a parser and interpreter for this language.

Integration. The feature prototypes were designed to allow us to develop, in isolation, the main features required for an evolution planning plug-in based on our approach. The final phase of development was an integration phase aimed at tying these features together into a unified tool.

For the most part, the project proceeded as expected. All the planned feature prototypes were completed. However, there were a number of unexpected challenges.

First, the MagicDraw API turned out to be less flexible than we had hoped, making it difficult or impossible to implement several of the features that we had envisioned. For example, the API does not provide much flexibility for plug-in developers to overhaul or replace the default user interface, which made implementing the kind of work flow we had envisioned impractical. I will discuss this further in section 6.2.2.

In addition to limitations in API *functionality*, we also experienced problems pertaining to API *discoverability*. Much of this was due simply to the significant size of the MagicDraw OpenAPI. With such a large API, it is difficult for a small team of part-time developers to learn it quickly. Thus, it was often difficult to find the features we needed, or even to know *which* features the API supported. Even when we found the relevant part of the API, the documentation was sometimes inadequate, so trial and error were required to understand a class's function.

These delays (along with technical difficulties in the project set-up phase) slowed our progress significantly. Although we completed all planned feature prototypes, many of them were not as fully fleshed out as we had anticipated, as mentioned above. This also left insufficient time for the integration phase to be satisfactorily completed. Thus, at the end of the master's studio project, the plug-in was more like an agglomeration of features than a unified tool with a consistent design.

6.2.2 Lessons learned

Our experience provided a number of insights about challenges in implementing an architecture evolution tool as a plug-in. Three aspects of the project that we found particularly challenging were understanding the kinds of variation supported by the underlying architecture modeling framework, exercising control over the modeling tool's user interface, and manipulating the visual presentation of the model. I now discuss each of these challenges in turn.

Understanding the kinds of variation that the framework supports Plug-ins are, by nature, subject to the limitations of the frameworks they are based on. A framework, as the name implies, establishes the boundaries within which plug-ins may operate and rules to which they must adhere, and a plug-in can subvert these boundaries and rules only with great difficulty. Therefore, when considering developing a plug-in to some framework, it is important to consider whether the intended features of the plug-in align with the intended use of the framework—whether the framework's variation points are appropriate to the needs of the plug-in. In this project, we found that some of the features we wished to implement (e.g., traversing and analyzing the evolution paths in a model) aligned well with the capabilities of the MagicDraw OpenAPI, while some others (e.g., ensuring the consistency of the evolution model) did not.

For a software architecture evolution plug-in, one especially important kind of variation is variation in the user interface, which I discuss next.

Controlling the user interface At the outset of the project, our hope was that it would be possible to present the user with custom interfaces for defining evolution paths, such as a "transition specification mode" in which users could select operators from a palette and visualize their effects on the architecture. This turned out to be infeasible. The MagicDraw OpenAPI is ideally suited to adding simple interface elements such as new toolbars and menus; it does not provide an easy way to define new user work flows with custom views of the system, or to overhaul the basic user interface.

For an architecture evolution plug-in to be usable, it is essential that significant modifications to the basic user interface of the modeling tool be possible. For example, to prevent the evolution model from becoming inconsistent, the plug-in must be able to prevent users from making arbitrary changes to the model. We found this to be impractical with the MagicDraw OpenAPI.

Manipulating presentational elements A significant practical challenge of developing an architecture evolution tool based on our approach—or any approach involving representation of intermediate states—is positioning presentational elements. Because intermediate states are generated from earlier states by means of operators, it is the plug-in's responsibility to ensure those newly generated states are rendered appropriately (e.g., by deciding where to position newly added components). A mature software architecture evolution tool might have to make use of fairly intelligent positioning algorithms to produce a satisfactory user experience. Ideally, we might hope that the architecture modeling platform on which our plug-in is based would provide advanced diagram layout features. MagicDraw provides only slight help in this area; although it has some automatic layout tools, they are quite limited.

The characteristics of the MagicDraw API that presented challenges for us—the rigidity of its user interface, the difficulty of prohibiting user actions that cause model inconsistencies, support for advanced diagram layout features—should certainly not be seen as failings or defects in MagicDraw. Rather, they arise from deliberate (and not unreasonable) API design choices that the developers of MagicDraw made in order to better support the core features that they wished to expose. But they do present limitations in the context of an architecture evolution modeling plug-in, and such limitations should be carefully considered by developers of any such plug-in.

It is also unlikely that the limitations we encountered are really particular to MagicDraw. I suspect that most architecture modeling tools will tend to share many of these same limitations. Thus, many of the challenges we encountered are likely to pose difficulties for anyone developing an architecture evolution tool as a plug-in to an existing architecture modeling tool. If that is the case, these challenges may be fairly fundamental ones that future developers of architecture evolution tools will be forced to confront (or they will have to take great care in selecting an architecture modeling tool that is closely aligned with the necessary requirements).

Of course, an alternative is to develop an architecture evolution tool from the ground up, without basing it on an existing architecture modeling framework, but this would involve forgoing the important benefits that a plug-in based approach provides, such as the ability to leverage the existing architecture modeling facilities that such a framework provides and the greater usability that results from plugging into a modeling framework with which users are already familiar.

Our approach to architecture evolution modeling is designed to lend itself to tool support, and many of the core features of our approach appear to be implementable in a reasonably straightforward way. But the implementation effort I have described here revealed a number of specific and significant engineering challenges that developers of future architecture evolution tools would be wise to consider carefully.

6.3 Automated generation of architecture evolution paths

The third and final implementation effort that I will describe is rather different from the first two. While the projects described in sections 6.1 and 6.2 were directed at developing prototypes that would illustrate a broad array of features of our approach, this third project was a narrower investigation of a single issue: the automatability of evolution path generation.[14]

[14]This work is described in a conference paper that we published this year [20], from which much of the material in this section is adapted. I would like to acknowledge here the contribution of Ashutosh

In motivating this thesis work in section 1.2, I painted a picture of how a hypothetical architect would use our approach, first defining the initial and target architectures, then carefully filling in the rest of the evolution graph by defining the intermediate states, then applying constraints and evaluation functions to analyze evolution paths. The prototype tools described in sections 6.1 and 6.2 assumed a similar work flow.

The problem with such an approach is that it imposes a substantial burden on the architect. The architect must explicitly define the candidate evolution paths and specify the evolutionary transitions that occur within each such path. In our original prototype, Ævol, this was done through explicit definition of intermediate states. The JPL case study and the follow-on MagicDraw plug-in work streamlined this process by allowing intermediate states to be defined in terms of the initial state by applying evolution operators. But even in this streamlined process, definition of intermediate states can be an onerous task in a scenario with many candidate evolution paths, and numerous transitions within each path.

A better approach would be to generate these evolution paths automatically. Rather than fully specifying the evolution space, the architect could simply define the initial and target architectures; then a tool could select architectural transformations from a predefined library of operators and apply them in sequence to generate candidate paths from the initial architecture to the target architecture.

While this would alleviate the burden on the architect, it introduces a new difficulty: determining how to compose the operators together so as to generate the target architecture from the initial architecture. (Given n operators, each with m parameters ranging over a domain of d architectural elements, there are $(nd^m)^l$ evolution paths of length l. Clearly an undirected brute-force search for an optimal path would be unwise.) This problem is very much akin to the *planning problem* in artificial intelligence [99]: given a description of the state of the world, a goal, and a set of actions, how can we generate a *plan*—a sequence of actions leading from the initial state to the goal?

In this section, I describe our attempt to apply existing approaches and tools from automated planning to the architecture evolution path generation problem. Adapting these existing approaches to software architecture evolution is a difficult problem, as it requires consideration of a number of concepts—architectural changes, technical and business constraints, rich temporal relationships among events, trade-offs among evolution concerns—that do not translate easily into the planning domain.

6.3.1 Automated planning

Given a set of states S, a set of actions $A : S \to S$, an initial state $s_0 \in S$, and a set of goal states $S_g \subseteq S$, the *planning problem* is the task of finding a sequence of actions that, when applied to s_0, yield one of the goal states.[15] The planning problem has broad applications, from robotics to business management to natural language generation, and has received a great deal of attention from artificial-intelligence researchers. A

Pandey, with whom I collaborated on this research.

[15]This is a very abstract formulation of the planning problem. For a discussion of alternative definitions, including some that are more computationally oriented, see Ghallab et al. [99].

variety of approaches and tools for solving planning problems have been developed over the last several decades.

To solve a planning problem, a planner must receive a specification of the problem in a standard format. A number of specification languages for planning problems have been devised, but by far the most popular—the lingua franca of automated planning—is the Planning Domain Description Language. PDDL was first introduced in 1998 [151] and soon became a de facto standard in the planning literature, facilitating reuse of research and allowing easy comparison of planners, systems, and models [87]. These qualities, along with its feature set, made PDDL a good choice for our work.

PDDL has undergone several revisions. The version that we adopted is PDDL2.1 [87], introduced in 2002, which greatly enhanced the language's expressivity by introducing:

- *numeric fluents*, which provided full support for modeling numerically valued resources such as fuel and distance;
- *durative actions*, which greatly enriched the temporal expressiveness of the language; and
- *plan metrics*, which allowed specification of a metric with respect to which a plan should be optimized (e.g., minimize fuel consumption).

All three of these are extremely useful for modeling architecture evolution problems, as we will see later. Most of PDDL2.1 is now reasonably well supported by the leading planners. There have subsequently been further additions to the language, such as the introduction of derived predicates in PDDL2.2 [78] and constraints and preferences in PDDL3 [97]. While these features would certainly have been useful to us, they are not as broadly supported by planners, so we chose to target PDDL2.1.

A PDDL specification comprises two parts, which appear in separate files: a *domain description* (consisting chiefly of a description of possible actions that characterize domain behaviors) and a *problem description* (consisting of the description of specific objects, initial conditions, and goals that characterize a problem instance). Thus, a domain description can be shared across multiple planning problems in the same domain. Both the domain file and the problem file are expressed in a Lisp-like syntax, as a list of parenthesized declarations.

In PDDL2.1, a domain file can declare:

- A set of *types* to which objects may belong. Each type may optionally declare a supertype. If a type does not declare a supertype, it is deemed to be a subtype of the built-in type *Object*; all types are ultimately subtypes of *Object* (perhaps indirectly). A type is simply a name; it does not define a set of properties or methods. Rather, predicates, functions, and actions can specify the types that they govern.
- A set of *predicates* over objects.
- A set of *functions* that map $Object^n \rightarrow \mathbb{R}$.

- A set of *action* schemata, each comprising a list of parameters, the conditions under which the action may be taken, and the effects of the action. A *durative action* additionally specifies its duration.

A problem file declares:

- A list of *objects*.
- The *initial conditions*, consisting of truth assignments for predicates and numeric value assignments for functions.
- The *goals*, which are defined in first-order predicate logic.
- A *metric* to be minimized or maximized.

A planner takes a domain description and problem description as input and produces a *plan* as output—a timed list of actions (with parameters specified) that achieves the specified goals.

6.3.2 Approach

The problem of generating an evolution path from an initial architecture to a target architecture can be framed as a planning problem in the sense of section 6.3.1 as follows:

- S, the set of states, is defined to be the set of legal software architectures.
- A, the set of actions, is defined to be the set of evolution operators.
- s_0, the initial state, is defined to be the initial architecture.
- S_g, the set of goal states, is defined to be the singleton set consisting of the target architecture of the system.

With the problem framed in this manner, we can apply automated planning tools to the task of generating evolution paths.

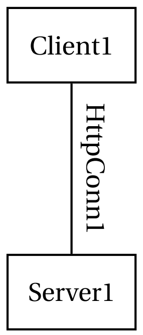
In the remainder of this section, I will describe how an architecture evolution problem can be translated into a planning problem expressed in PDDL. (A summary appears in table 11.) Section 6.3.3 will make this discussion concrete by showing how we applied it to a specific architecture evolution problem and used off-the-shelf planners to generate evolution paths.

Representing the initial and target architectures. The first step of modeling an architecture evolution problem is to specify the initial and target architectures. Recall from section 2.1 that a software architecture is conventionally conceived as an arrangement of architectural elements such as components and connectors. In addition, as we have seen, these components and connectors are often expressed in terms of *component types* (such as *WebService* or *Database*) and *connector types* (such as *EventBus* or *HttpConnection*).

PDDL's type system, though simple, is quite adequate for our needs. We can define component and connector types as types in the PDDL domain description, then define the components and connectors themselves as PDDL objects of the defined types. Finally, the relationships among the components and connectors

Evolution element	PDDL translation
Transitional architecture	State
Architectural element type	Object type
Architectural element	Object
Relationship among architectural elements	Predicate
Evolution operator	Action
Parameter	Action parameter
Precondition	Action condition
Architectural transformation	Action effect
Property	Action duration, or action effect modifying function value
Evolution path	Plan
Initial architecture	Initial state
Target architecture	Goal state
Path constraint	PDDL3 constraint, or action condition supported by predicates to track the state
Path evaluation function	Metric

Table 11. *Summary of our approach for translating elements of an architecture evolution problem into PDDL.*



Domain description

```
(:types (Component Connector - Object
    Client Server - Component
    HttpConnection - Connector))
(:predicates
    (connects ?c - Connector ?a ?b - Component))
```

Problem description

```
(:objects (HttpConn1 - HttpConnection
    Client1 - Client
    Server1 - Server))
(:init (connects HttpConn1 Client1 Server1))
```

Figure 16. *An extremely simple software architecture and its PDDL representation.*

can be expressed using predicates, which are defined in the domain description and assigned truth values in the problem description. Figure 16 shows a simple example.

The specification of the initial architecture will appear within the *:init* block, which defines the initial conditions, and the specification of the target architecture will appear within the *:goal* block, which defines the goals.

Representing evolution operators. An evolution operator, of course, corresponds to a PDDL action. But how can we actually capture an evolution operator as an action using the specification facilities that PDDL provides?

Let us consider each of the parts of an evolution operator (as described in section 2.2) in turn. An operator comprises:

- A set of *parameters*. For example, a *wrap legacy component* operator will take as a parameter the component to wrap. In PDDL, an action likewise specifies its parameters.
- A description of the *architectural transformations* that the operator effects. These are expressed as a sequence of elementary architectural changes such as *delete component* or *attach connector*. In PDDL, we can represent these transformations via the action's effects.
- A description of the operator's *preconditions*. These map into PDDL in a straightforward manner; in PDDL, any action may declare its preconditions in terms of predicates and functions over the action parameters.[16]

[16] For durative actions, this is generalized to include other kinds of conditions—not only preconditions (conditions that must hold at the start of an action), but also conditions that must hold at the end of an action, or over its entire duration. These are specified with the temporal annotations *at start*, *at end*, and *over all*. (These can also be applied to effects.)

- A list of *properties* of the operator, used to support evaluation functions. Examples of properties are the time needed to carry out the operator, the cost of doing so, and the operator's effects on system performance. In PDDL, the duration property is given special prominence due to its importance in temporal reasoning; a durative action must specify its duration. As for the other properties, these are best captured via PDDL effects. For example, if an evolution operator has a cost property (indicating that it costs $1,000), we can define a *cost* function in the PDDL specification, then add an "increase cost 1000" effect to the action.

One subtlety worth noting is that PDDL does not permit actions to create new objects (nor destroy existing ones). This is significant because many evolutions entail the creation of new architectural elements, or the decommissioning of existing ones. As a result, in an evolution that may involve creation of new elements, we must declare some *potential* objects that do not exist in the initial architecture but may be used to stand in for elements created during the evolution. In this case, we can define an *isReal* predicate that is false for such potential objects and becomes true when an action creates a new architectural element out of a potential object. Such approaches have substantial limitations and are rather cumbersome, and Frank et al. [89] identify this as an important limitation of PDDL. (A related point is that a PDDL specification can have only finitely many objects, while the set of software architectures reachable via a set of evolution operations may be infinite in general.)

Representing path constraints. Path constraints are perhaps the most challenging element of an architecture evolution problem to translate into PDDL. As section 2.4 explained, path constraints can be naturally represented using temporal logic. In chapter 3, I described an extension of linear temporal logic that is particularly suitable for expressing evolution path constraints. Unfortunately, PDDL2.1 does not have any means to define constraints using temporal logic.

One way of addressing this would be to develop a way of translating temporal formulas into PDDL directly. Indeed, there is previous work in this direction; Cresswell & Coddington [61] present a means of compiling an LTL goal formula into PDDL. They use a two-step process; first they generate a finite-state machine that accepts traces of the LTL formula, then they encode this automaton as a collection of facts in PDDL and modify the actions to track the current state. This process is conceptually complex and encumbers the specification with numerous state variables. Therefore, I leave to future work the challenge of extending this compilation process to the augmented version of LTL that we use to capture path constraints.

To avoid such conceptual complexities in this work, we took a pragmatic approach: we characterized certain restricted classes of path constraints (with an eye toward the kinds of constraints that will arise in the example of section 6.3.3) and showed how they could be easily represented using the existing facilities of PDDL.

- *Constraints that must hold throughout an evolution.* The simplest possible kind of constraint is one that must hold continuously through the entire duration of the evolution (e.g., a system must always be protected by a firewall, or a trusted component may never connect directly to an untrusted one). In LTL, such a constraint takes the form $\Box\phi$ for some propositional formula ϕ. Despite their simplicity, these constraints are quite common. Such a constraint amounts to an architectural constraint that persists through an evolution. In PDDL, we can model such a constraint easily (if verbosely) as a condition on every action.

- *Ordering constraints.* Another common class of constraints comprises constraints that govern the order of the operations that are to be carried out in the course of an evolution. For example, a firewall must be installed before connections to a protected resource are permitted; a high-priority client should receive a service upgrade before a low-priority one. Such constraints are also generally easy to model in PDDL. If an operator B must be preceded by an operator A, then we can have action A set a predicate, *aExecuted*, that is a precondition for operator B.

- *Timing constraints.* Constraints on the time at which evolution operations are carried out, or the time by which certain goals must be achieved, are extremely common in real-world evolution. In the simplest case, there may be a requirement that the evolution be completed by a specific date. In more complex cases, there may be a set of such requirements: feature A must be available for client 1 by April, feature B for client 2 by July, and so on. These can be modeled in PDDL by setting appropriate conditions on durative actions.

  A more complex kind of timing constraint is a constraint that certain actions can be performed only at certain times. A real-world example is that many retailers, such as Amazon.com and Costco, refrain from making major software changes during the Christmas shopping season, so as not to introduce bugs during a period of heavy use. In section 6.3.3, we will see another example, in which certain operations can be carried out only on certain days of the week. These are also expressible in PDDL. There are some challenges, however, which are explored in section 6.3.3.

There are many constraints that do not fit into these categories, but it seems that many of the constraints that arise in real-world evolutions do fall into these groups (see, for example, section 5.4.3). In section 6.3.3, we will see how various constraints can be represented in PDDL.

A final point to note is that PDDL3 has its own notion of a constraint. Like our constraints, PDDL3 constraints express conditions that must be met by an entire plan (in contrast with conditions in PDDL2.1, which are evaluated locally, with respect to a particular point in time). Moreover, these constraints are expressed in a syntax reminiscent of temporal logic, with operators such as *always*, *sometime*, *at-most-once*, and so on. However, there are substantial restrictions; most significantly, these modalities may not be nested. As a result, this constraint language is less expressive than LTL. Nonetheless, PDDL3 constraints would be a useful way of expressing a broad class of evolution path constraints. However, because we targeted PDDL2.1, I do not discuss them further here.

Representing path evaluation functions. As explained in section 2.5, an evaluation function provides a quantitative evaluation of a path. As we have seen, there may be evaluation functions for various dimensions of concern, such as cost and availability, which can be composed together into an evaluation function that captures a notion of *overall path utility*.

All of this can be translated into PDDL. Evaluation functions such as cost and availability can be modeled as nullary functions in PDDL, and their values can be modified by actions as appropriate. Finally, we can use these values to set a *plan metric* in the problem description, which planners will try to optimize in generating a plan. This metric can simply be a reference to a function, or it can be an arbitrary arithmetic expression. The metric can also incorporate the total duration of the plan by using the built-in variable *total-time*.

6.3.3 Application

To show how this approach could be used in practice, and to provide a demonstration of its applicability, we applied it to an evolution scenario. The scenario is based loosely on a real-world data migration experience that we had previously elicited (for other purposes) from a practicing software engineer. We elaborated this experience into a complete description of an architecture evolution problem, so that it would be specific enough to operationalize as a planning problem. Then, using the approach described above, we translated this scenario into PDDL.[17] Finally, we used two different off-the-shelf planners to generate plans and evaluated the results.

Our example is based loosely on a real-world data migration scenario, in which a company had to migrate a number of services from an old data center to a new data center. The planning for this migration was nontrivial, because there were a number of interacting constraints governing how the various services had to be moved. For example:

- Different services had different kinds of availability requirements. For example, some services had to be continuously available for regulatory reasons (zero planned downtime). In other cases, there were periods when certain services were required to be online (e.g., the payroll system had to be online at the end of each payroll period).
- Different services had to be moved in different ways. Some services (particularly those hosted on Unix systems) could be easily cloned into the new data center using the corporate storage area network. Other services were more finicky and could not be cloned automatically; manual intervention was required to migrate these services. And there were a few unique legacy services that were running on custom-built, special-purpose hardware. These services were so closely tied to the machines on which they were running that the only practical way to migrate them was to load the machines onto a truck and drive them to the new data center.

[17]For the sake of replicability, I have reproduced the entire PDDL specification in appendix C.

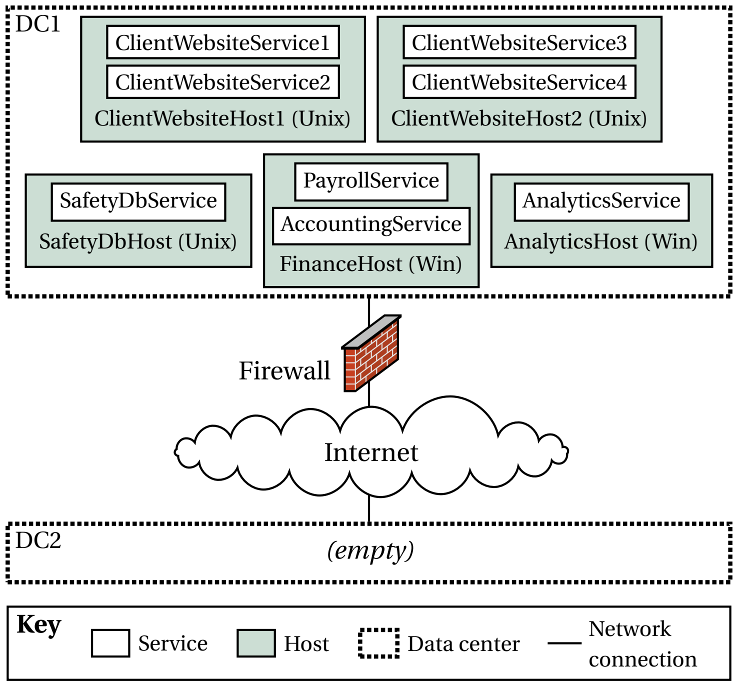


Figure 17. *Deployment view of the initial architecture of the data migration scenario.*

- No services could be established in the new data center until a firewall was installed there.

In the real-world experience on which our scenario was based, the architects experienced significant difficulty in managing these interacting constraints to develop a satisfactory plan. The planning process ultimately took roughly six months, and the migration itself was carried out over several weekends.

We elaborated this scenario by adding additional architectural details as necessary to create a complete specification of an architecture evolution problem. For example, although we had general information about the kinds of architectural elements and evolution constraints, we did not have a list of specific service names and locations, so we invented fictitious service names and assigned them to hosts at will.

The initial architecture is shown in figure 17. There are five hosts in data center DC1, each with one or more services, all of which must ultimately be migrated to DC2. We defined a number of specific evolution constraints based on the real-world constraints above. For example, we specified that the payroll service in figure 17 must be available on Mondays to permit payroll processing, and we defined rules governing how the services could be moved (e.g., Unix services can be cloned to a new data center over the network, but the analytics engine is tied to special-purpose hardware that must be physically relocated). We defined six evolution operators: *install network switch*, *install firewall*, *decommission host*, *clone host*, *manually transfer service*, and *physically relocate host*. Finally, we defined two ways of evaluating path quality: cost and duration. *Cost* is affected by when and how systems are migrated (migrating a system on weekends is more expensive than during normal working hours, and physically moving a host is much more expensive than cloning a host over the network). *Duration* refers to the overall time to complete the evolution.

Representing the initial and target architectures. We represented the initial architecture following the approach described in section 6.3.2. In the domain description, we defined PDDL types for the architectural element types: *DataCenter*, *Service*, and *Host* (with subtypes *UnixHost* and *WindowsHost*). We defined predicates to indicate relationships among elements, such as an *is-in* predicate that holds when a given host is in a given data center and an *is-on* predicate that holds when a given service is on a given host. With these types and predicates defined, we were able to translate the initial architecture in figure 17 into a PDDL description of the initial state.

Representing the target architecture as a set of goal conditions, on the other hand, entails some subtleties. In principle, we could define the target architecture by the same method that we defined the initial architecture—specify exactly which services are on which hosts and which hosts are in which data center. In practice, however, this would be a bit too restrictive. Because services can be migrated in multiple ways—cloning, manual service-by-service migration, or physical relocation—there are actually multiple legal end states. For example, we could clone ClientWebsiteHost1 onto a new host in DC2 and decommission ClientWebsiteHost1, or we could instead move ClientWebsiteHost1 itself to DC2. These would result in slightly different end states, but either is permissible from the standpoint of path correctness; the choice should be left to the planner. Thus, we defined the goals of the evolution in more general terms; we defined a permissible end state to be one in which (1) all services end up in DC2 and (2) no hosts remain in DC1.

In principle, these goals are easy to represent in PDDL:

```
(:goal (and
    ; All services end up in DC2.
    (forall (?s - Service)
        (exists (?h - Host) (and (is-on ?s ?h) (is-in ?h DC2))))

    ; No hosts remain in DC1.
    (not (exists (?h - Host) (is-in ?h DC1)))))
```

Unfortunately, practical considerations prevent such a straightforward approach. Many planners—including OPTIC, one of the planners we used in this work—do not support goals with negative or existential operators. To get around this, we defined helper predicates such as *was-migrated* (to indicate that a service has been migrated) and *was-removed-from* (to indicate that a host has been removed from a data center) and modified the actions to update them throughout the evolution. We then declared our goals as follows:

```
(:goal (and
    ; All services end up in DC2.
    (forall (?s - Service) (was-migrated ?s))
```

```
(:durative-action manuallyMigrateService
    :parameters (?s - Service ?h1 ?h2 - Host ?d - Day)
    :duration (= ?duration 3.9)
    :condition (and
        (at start (is-on ?s ?h1))
        (over all (is-in ?h1 DC1)) (over all (is-in ?h2 DC2))
        (over all (has-firewall DC2))
        (over all (network-switch-installed))
        (at start (not-yet-migrated ?s))
        (over all (can-be-migrated-individually ?s))
        (over all (ok-to-move-on ?s ?d))
        (at start (no-work-in-progress))
        (over all (today ?d))
        (over all (>= (allowed-downtime ?s) 3.9))
        (at start (<= time-since-last-day 4.1)))
    :effect (and
        (at end (is-on ?s ?h2))
        (at end (was-migrated ?s))
        (at end (not (not-yet-migrated ?s)))
        (at end (not (is-unused ?h2)))
        (at start (not (no-work-in-progress)))
        (at end (no-work-in-progress))
        (at start (increase (total-cost) (* 20 (cost-multiplier ?d))))
        (at end (increase current-hour 3.9))
        (at end (increase time-since-last-day 3.9))))
```

Figure 18. *Expression of an evolution operator in PDDL.*

```
        ; No hosts remain in DC1.
        (was-removed-from ClientWebsiteHost1 DC1)
        (was-removed-from ClientWebsiteHost2 DC1)
        (was-removed-from SafetyDbHost DC1)
        (was-removed-from FinanceHost DC1)
        (was-removed-from AnalyticsHost DC1)))
```

This complicated the domain description, but it allowed us to express our goals crisply despite the limitations of planners.

Representing the evolution operators. We represented the operators as actions in accordance with the approach described in section 6.3.2. Figure 18 shows an example: the action for manual migration of a service from one host to another. Much of this is straightforward. The action first defines its parameters: the service being migrated, the hosts it is moving from and to, and the current day (I will explain this parameter shortly). Then it defines its duration: 3.9 hours.

Many of its conditions correspond directly to preconditions of the evolution operator. For example, to migrate a service s from host h_1 to host h_2, clearly s must be on

h_1 at the outset. We also require that h_1 is in DC1 and h_2 is in DC2 (we only want to move services from DC1 to DC2), and we require that the firewall and network switch are already installed.

The conditions that reference *no-work-in-progress*, *today*, and *time-since-last-day* are used in modeling the passage of time, which I describe later in this section.

Many of the effects are straightforward specifications of the evolution operator's architectural transformations. When the *manuallyMigrateService* operator is applied, the effect on the architecture is that service s is now on host h_2. We also must set here a number of helper predicates, as mentioned earlier, such as *was-migrated*, *not-yet-migrated*, and *is-unused*. The effects that mention *no-work-in-progress*, *current-hour*, and *time-since-last-day* are used to support the modeling of the passage of time and will be discussed under the next subhead. Finally, the effect that increases *total-cost* is used for cost optimization, which I will describe later in this section.

Representing time. The most difficult part of representing this scenario in PDDL was capturing its temporal aspects. The temporal features that PDDL provides fall well short of this scenario's needs. In particular, this scenario (like many evolution problems) is steeped in references to real-world time—that is, clock time, or calendar time. The payroll service must be available on Mondays; the accounting service can be moved only on weekends; operations are most expensive when carried out on weekends. PDDL is ill suited to representing such considerations. PDDL's conception of time is a continuous timeline, extending from zero to infinity. To reason about concepts such as Mondays and working hours, we must model them ourselves—and do so in a way consistent with PDDL's own model of time. This is rather difficult.

In this scenario, we are interested only in working hours; the company, in this scenario, has only day employees, and all work takes place between 9 a.m. and 5 p.m. We therefore interpret PDDL's timeline within an eight-hour day cycle; time indices between 0 and 8 represent Monday, times between 8 and 16 represent Tuesday, and so on.

This simplifies the specification because we do not need to model the empty nighttimes. However, it also creates some difficulties. We now must prohibit actions from spanning day boundaries; we do not want the planner to schedule a four-hour action as beginning at time 6 (Monday at 3 p.m.) and ending at time 10 (Tuesday at 11 a.m.). An action must be completed within a single day. Enforcing this rule in PDDL is difficult.

First we need a way to keep track of time. In PDDL, an action does not know when it is occurring; that is, it has no way to refer to the current time. If we want to keep track of time, then, we must do it ourselves. To do so, we define a nullary function, *current-hour*, and we add to each action an effect to set its value:[18]

```
(at end (increase current-hour ?duration))
```

[18]As can be seen in figure 18, we actually hard-code the duration rather than using the *?duration* parameter, because some planners have trouble when the *?duration* parameter appears in effects and conditions.

If PDDL provided a sufficiently rich set of arithmetic operators, this alone would be sufficient to prevent actions from crossing day boundaries; each action could have a precondition

$$?duration + (current\text{-}hour \bmod 8) \leq 8.$$

Unfortunately PDDL does not have a modulo operator. Instead we must further complicate the specification with a *time-since-last-day* function. As with *current-hour*, every action has an effect that increments this value by the action's duration. We also create a special action, *waitTillNextDay*, that waits until the next multiple of 8 and resets the value of *time-since-last-day* to 0. Finally, we give each action a precondition that prevents it from being scheduled when there is less time remaining in the day than is required to carry it out:[19]

```
<= time-since-last-day (- 8 ?duration)
```

Days of the week pose yet another challenge. Recall that some services can be moved only on certain days; for example, the accounting service can be moved only on weekends. Again, with a modulo operator, this would be easy; weekends are those times such that $40 \leq current\text{-}hour \bmod 56 < 56$. Since PDDL lacks a modulo operator, we must again complicate the domain description, this time by defining a *Day* type (with values *Monday*, *Tuesday*, etc.) and a predicate over days, *today*, that indicates the current day. We then modify the *waitTillNextDay* action to set this predicate. This will permit us to express constraints pertaining to days of the week.

A final temporal rule that we enforced was to forbid concurrency. This scenario describes a single, small team of engineers evolving a simple information system; they can work on only one thing at a time. PDDL2.1, on the other hand, is based on an action execution model that is concurrent by default; a planner will gladly schedule all actions to occur simultaneously at time 0 if allowed to do so. To prevent this, every action has a condition that prevents it from executing when the *no-work-in-progress* predicate is true; every action sets this predicate to false at the beginning of its execution and resets it to true at the end.

Representing the path constraints. We have already seen the representation of some of the constraints in this scenario. The prohibition on concurrency, for example, is achieved by means of action conditions and effects. The requirement that a firewall must be installed before any services are migrated is similarly simple to model via a *has-firewall* predicate that is set by the *installFirewall* action and appears as a precondition for all the migration actions.

The availability constraints were more challenging to model. As we just saw, a substantial infrastructure is required to model days of the week in a way that can support the expression of these constraints. With this infrastructure in place, we can specify which services may be moved on which days by defining a predicate, *ok-to-move-on*, over services and days, and setting its values in the problem description (e.g., "ok-to-move-on AccountingService Saturday"). Then, we set on each migration action a condition that the given service may be moved on the current day. To do so,

[19] Again, due to planner limitations we actually hard-code the duration.

we add a parameter *d* of type *Day* to the action to represent the current day (which we enforce with a “today ?d” condition) and then add the condition “ok-to-move-on ?s ?d” (see again figure 18 for a full example).

We use a similar strategy to define the constraints governing how different services may be migrated. For example, to define which services can be manually migrated over the network, we define a *can-be-migrated-individually* predicate over services, which is a condition of the *manuallyMigrateService* action.

Most of these constraints are specified using the same general idiom: the constraints themselves often appear as action conditions, but they often are supported by predicates that keep track of state (which is maintained through the use of action effects). This is an ad hoc version of the kind of state-based reasoning that would occur if we were to adopt a more formal means of translating constraints expressed in temporal logic into PDDL à la Cresswell & Coddington, as suggested in section 6.3.2.

Representing the path evaluation function. In this evolution, the goal is to minimize cost; thus we have a single evaluation function, which we model in PDDL by the nullary function *total-cost*. The value of this function is incremented by the actions, and the function is defined as the goal metric in the problem description.

The main complication is that the costs of actions are not fixed. Actions are more expensive on weekends than during normal working hours. The straightforward way to model this would be with conditional effects. Unfortunately they are not well supported (by now a familiar refrain).

Instead we introduce a *cost-multiplier* function over days, which we value at 1 for weekdays and 3 for weekends. Since each action has a parameter representing the current day of the week (see section 6.3.3), each action can incorporate this cost multiplier in its effect on *total-cost*, as shown in figure 18.

Generating an evolution path. The final PDDL specification was of moderate size: 24 objects, 14 predicates, 8 functions, 130 initial conditions, and 9 durative actions (each with, on average, 8 conditions and 9 effects). With the specification complete, the next step was to generate a plan.

We used two different planners to demonstrate a key advantage of PDDL: its status as a lingua franca supported by many planners. We chose LPG-td [98] and OPTIC [26] as the two planners due to their feature sets, ease of installation and use, maturity, planning quality, and general good reputation.

Both planners work by first attempting to generate a correct (but possibly low-quality) plan, then progressively refining the plan to improve its quality. Both planners correctly interpret our PDDL specification, find a correct solution within a few seconds, and refine it into an optimal solution soon thereafter.

Figure 19 shows an optimal plan generated by OPTIC. Observe that services are always moved by the cheapest means permissible—cloning is preferred, with manual migration and physical host transfer used only when required. In addition, the planner avoids scheduling any unnecessary activity on weekends; the only service migrated on the weekend is the accounting service, which is forbidden to be moved during weekdays.

```
0.000: (installswitch monday) [1.900]
1.901: (installfirewall dc2 monday) [0.900]
2.802: (clonehost2 clientwebsitehost2 unusedunixhost1 clientwebsiteservice3 clie
4.703: (clonehost2 clientwebsitehost1 unusedunixhost2 clientwebsiteservice1 clie
6.604: (waittillnextday monday tuesday) [1.399]
8.004: (physicallymovehost1 analyticshost analyticsservice tuesday) [5.900]
13.905: (clonehost1 safetydbhost unusedunixhost3 safetydbservice tuesday) [1.90
15.806: (waittillnextday tuesday wednesday) [0.201]
16.008: (decommissionhost safetydbhost dc1 wednesday) [3.899]
19.908: (decommissionhost clientwebsitehost2 dc1 wednesday) [3.899]
23.809: (waittillnextday wednesday thursday) [0.200]
24.011: (decommissionhost clientwebsitehost1 dc1 thursday) [3.899]
27.911: (manuallymigrateservice payrollservice financehost analyticshost thursda
31.812: (waittillnextday thursday friday) [0.200]
32.013: (waittillnextday friday saturday) [8.000]
40.015: (manuallymigrateservice accountingservice financehost analyticshost satu
43.915: (waittillnextday saturday sunday) [4.100]
48.016: (waittillnextday sunday monday) [8.000]
56.017: (decommissionhost financehost dc1 monday) [3.899]
```

Figure 19. *Output from OPTIC showing an optimal evolution plan. In bold are action names, which are followed by the action parameter assignments. At the beginning of each line is the time at which the action is executed.*

To reliably quantify the planners' performance on our specification, we ran our specification on each planner ten times on an Amazon EC2 medium instance (which has 3.75 GiB of memory and processing power roughly equivalent to a single 2.2-GHz core). Conducting multiple runs was particularly important because LPG-td's plan generation is highly nondeterministic; the initial plan is created based on a random seed.

In all ten runs, OPTIC was able to find a correct plan within 8 seconds and an optimal one (i.e., one that achieves the minimum possible cost) within 10 seconds. LPG-td was much slower at finding an optimal plan (unsurprisingly, since it is a much older planner than OPTIC), but it did succeed consistently within a few minutes, and it always found a correct, nonoptimal solution very quickly. Table 12 presents summary statistics.

We also ran a modified version of the problem in which we asked the planners to minimize plan duration and ignore cost. Times for these runs appear in the lower part of table 12.

6.3.4 Findings

Our experience demonstrated the viability of using automated planning tools to solve architecture evolution problems, but it also revealed challenges. I now discuss our main findings.

	Time to generate initial solution			**Time to discover optimal solution**		
	Min	Median	Max	Min	Median	Max
Minimizing cost						
LPG-td	3.6	3.8	3.9	40.6	126.2	278.4
OPTIC	6.6	6.6	7.9	7.6	7.6	9.3
Minimizing time						
LPG-td	3.7	3.7	4.1	49.8	90.5	287.4
OPTIC	6.6	6.6	7.8	15.3	15.4	17.4

Table 12. *Time to generate an evolution path. All figures are in seconds and are calculated over ten runs.*

PDDL is expressive enough to capture the significant concerns of an evolution problem. Despite some challenges, we were able to capture the evolution scenario in its entirety. The PDDL model of a planning problem is broadly consistent with a path-oriented view of architecture evolution; the correspondence between evolution operators and PDDL actions, for example, is satisfyingly direct.

We did have to contend with some limitations of PDDL. Most significantly, PDDL's simplistic model of time makes it difficult to specify constraints based on calendar time or clock time. Modeling constraints about which actions could occur on which days of the week, for example, posed significant challenges.

There has, incidentally, been considerable research on improved methods for expressing temporality in planning problems [65; 197], and languages have been developed that increase the temporal expressivity of PDDL [86; 146]. However, PDDL is far more widely supported than any other planning language.

Automated planners can effectively and efficiently generate evolution paths. Both automated planners we tried were able to quickly generate high-quality solutions to a moderately complex architecture evolution problem. This kind of automated path generation has the potential to ameliorate one of the most significant burdens of a path-based approach to software architecture evolution: the need for the architect to manually specify the evolution graph in full detail before beginning analysis. By taking advantage of automated planners, we are able to capitalize on decades of research in artificial intelligence, which allows paths to be generated quickly and intelligently.

Of course, we should be cautious about overgeneralizing based on a single experience. Some architecture evolution problems may be more amenable to solution by automated planners than others. More work is needed to evaluate the scalability and applicability of this approach.

Current off-the-shelf planners have limited feature sets. Although PDDL provides a powerful array of features for specifying complex and intricate planning problems, few (if any) existing planners support the language fully. In section 6.3.3, I mentioned numerous instances where we had to adapt our specification to accommodate the limitations of planners. In most cases, we were able to circumvent these limitations. That is, in many cases, planner limitations do not reduce the practical expressivity of the specification language, but they do make specifications more verbose and awkward. For example, poor support for negative and existential conditions forced us to clutter our specification with many otherwise unnecessary declarations.

This somewhat compromises PDDL's effectiveness as a lingua franca. Ideally, we should be able to write a PDDL specification once and use any PDDL-based planner to analyze it. In practice, planners' limitations are so idiosyncratic and poorly documented that adapting a specification to work with a particular planner is a frustrating and time-consuming process.

Debugging planning specifications is difficult. PDDL specifications are inherently difficult to debug. If a specification author forgets, for example, to specify a necessary initial condition in the problem description, causing the problem to be unsolvable, the planner will simply say that it is unable to generate a plan. There is no good way to track down the cause of the problem.

The experimental nature of available off-the-shelf planners exacerbates this problem. Even the most stable planners are fairly buggy and have limited documentation. We chose LPG-td and OPTIC for their relative maturity and stability, but while using them, we encountered many bugs and undocumented limitations. Error messages were often unhelpful, and it can be difficult to tell whether a problem is caused by a specification error, a limitation of the planner, or an outright bug. When a planner encounters something its designers did not anticipate, it is as likely to crash with a segmentation fault as it is to display any useful explanation of the problem.

6.4 Summary

This chapter is a compilation of three projects related to the topic of developing practical software architecture evolution tools. I have grouped these three pieces of work together in the same chapter because all three of them are smallish projects focusing on issues of architecture evolution tool implementation; thus all three of them are relevant to research claim 3 in section 1.3.3, the claim of implementability. However, the three projects had very different purposes and goals. In this summary, I compare the three pieces of work described in this chapter and explain their relevance to my thesis (and in particular the research claim of implementability). This discussion is summarized in table 13.

The first section in this chapter described Ævol, an early demonstration of our approach with which I was only peripherally involved (section 6.1). Thus, I do not consider it part of my thesis work, and I have not described it in any significant detail in this chapter. Moreover, Ævol was intended as an illustration of our approach and "a platform for exploring the foundations of architecture evolution" [96, p. 591], not an evaluation of implementability. I have referenced this early implementation work in this chapter primarily for the sake of completeness, but it provides only weak support for the research claim of implementability.

Section	Section 6.1	Section 6.2	Section 6.3
Implementation	AcmeStudio plug-in	MagicDraw plug-in	PDDL model
Motivation	Early research demonstration	Follow-on to JPL case study, focused on implementation issues	Ameliorate burden of defining evolution paths
Approach	Implement basic concepts from our approach in a plug-in to a research tool	Implement basic concepts from our approach in a plug-in to a commercial modeling tool	Adapt research on automated planning to generate evolution paths automatically
Findings	No formal findings were described in the publication; project was intended as a demo rather than an evaluation	Detailed discussion of issues and challenges in implementing an evolution tool as a plug-in to an existing architecture modeling framework	Evolution paths can be generated automatically using automated planning techniques, although there are practical challenges in using current planning tools
Relevance to thesis	Provides weak support for claim of implementability	Provides detailed treatment of practical challenges and issues in implementing architecture evolution tools	Demonstrates automatability of evolution path generation
Publication	[96]	[18]	[20]

Table 13. *A summary of the implementation projects described in chapter 6 and the differences among them.*

On the other hand, the second piece of work, the MagicDraw plug-in prototype (section 6.2), was intended explicitly as an examination of practical issues involved in implementing a software architecture evolution tool as a plug-in to an off-the-shelf architecture modeling framework. The findings of this work directly address the research claim of implementability by reporting and discussing a number of general challenges to architecture evolution tool implementation.

The final piece of work described in this chapter was an attempt to use research on automated planning to support automatic generation of evolution paths, based on specifications of an initial architecture, a target architecture, a set of operators, a set of constraints, and an evaluation function to optimize. This work was narrowly focused on the question of the automatability of the approach, as opposed to more general implementation issues.

7 Related work

Software evolution is a major area of research, and it has been for some time. Section 7.1 gives a brief overview of the extensive literature on software evolution generally and discusses why much of it is inadequate for addressing the problem of *architecture* evolution. Another relevant topic is software project planning. Section 7.2 summarizes work in this area and shows how it falls short of addressing the problem considered here. Section 7.3 discusses research on software evolvability. This work provides guidance on designing highly evolvable architectures and methods for evaluating architecture evolvability. Finally, there is now a growing body of research on the topic of planning software architecture evolution, which section 7.4 discusses in detail. Section 7.5 concludes by discussing how this thesis research relates to and distinguishes itself from related work.

For alternative views of the software architecture evolution research literature, see the work of Breivold et al. [38] and Jamshidi et al. [112], both of whom have undertaken systematic literature reviews on the topic of software architecture evolution.

7.1 Software evolution

Since the early days of software engineering there has been concern for the maintainability of software, leading to concepts such as modularization in support of maintainability [179], indications of maintainability such as coupling and cohesion [11; 227], code refactoring [175], and many others [100]. These techniques, which focus on the code structures of a system, have led to numerous advances, such as language support for modularization and encapsulation, analysis of module compatibility and substitutability [46], and design patterns that support maintainability [92]. While such advances have been critical to the progress of software engineering, they generally do not treat large-scale reorganization based on architectural abstractions. They focus primarily on the code level and do not capture information about architectural structure.

Some of the best-known foundational work in the area of software evolution is that of Lehman [135; 136], who in 1980 described (and in 1996–1997 revisited [137; 138]) a set of "laws of program evolution" based on software development experiences in the late 1960s and 1970s. Lehman's work was particularly notable because it dealt with the evolution of systems, not just the evolution of code. In this sense, Lehman's work was a forerunner of the later research on software architecture evolution described in sections 7.3 and 7.4.

Lehman's 1980 papers enunciated five laws:

1. **Continuing change.** A program either undergoes continuing change or becomes progressively less useful. Change or decay continues until it becomes

more cost-effective to replace the program with a recreated version.

2. **Increasing complexity.** As a program evolves, its complexity increases and structure deteriorates unless work is done to reduce it.
3. **The fundamental law.** The evolution process is self-regulating, with statistically determinable trends and invariances.
4. **Conservation of organization stability.** The global activity rate in a program evolution project is statistically invariant.
5. **Conservation of familiarity.** The release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

The first two laws have particularly clear relevance here. Systems that fail to evolve become less useful over time; and systems that evolve but fail to do so in a principled fashion become less maintainable and more complex over time. These truths, which remain relevant today, justify and explain the need for principled, high-level evolution to keep systems relevant and coherent.

7.2 Software project planning

In the domain of project planning, traditional project management approaches and software development planning approaches provide ways to plan and analyze software development. Broadly speaking, these techniques address the same high-level problem as this dissertation: planning the best way to carry out major changes to a software system. However, the approach and applications are quite different.

Among the most well-known and well-developed approaches for project planning are cost and effort estimation techniques such as COCOMO [31], the Putnam model [186], SEER-SEM [82], estimation by analogy [204], dynamics-based techniques such as that of Abdel-Hamid [1], and others [33]. However, such techniques focus primarily on the end state of a maintenance or development effort. They tend to be more concerned with gauging the overall cost of an evolution (or other software project) rather than developing a detailed plan of evolution or reasoning about sequences of developments. The approach presented here, on the other hand, has the explicit goal of selecting a particular path of evolution—one that optimizes utility subject to relevant technical and business constraints.

Aside from these approaches to cost modeling, there is a great deal of general work on software project management and planning. Boehm [32] describes good elements of a project plan. Phillips [183] describes the planning process and the elements of a software project plan. Conger [55] describes steps for developing a software project plan. Wu & Simmons [225] present a tool-based approach for developing and managing software project plans. Software project planning is also a key process area in the Capability Maturity Model, which provides guidance on what constitutes a mature project planning process [180]. While these resources provide useful general guidance on effective project planning, they do not provide architects with domain-specific, situationally tailored guidance on how to structure an evolution.

Finally, none of these approaches directly address the architecture-level problem. While they may provide general guidance on the way project planning should occur, or the amount of effort that is likely to be necessary to complete the evolution, they do not provide software architects with guidance on what operations must be carried out, what order they should be done in, and what alternatives exist, nor are they capable of stating and enforcing constraints on a system's architectural structure. Project planning and software architecture evolution modeling complement each other; project planning approaches can be used to get a rough cost estimate of an evolution project and figure out what resources to allocate to it, then architecture evolution tools can be used to plan the architectural operations necessary to achieve the evolution goals.

7.3 Evaluating and architecting for evolvability

Architectural treatments of evolvability can be roughly divided into two classes: (1) guidance on designing highly evolvable software, and (2) methods for evaluating the evolvability of an architecture. The first class of research answers the question of how to architect software that will be easy to evolve or adapt in the future. The second answers the question of how to determine the ease with which a software system can be evolved. Thus, the first is prescriptive and the second is descriptive. Of course, these two treatments are closely related; often, the reason one wants to evaluate the evolvability of an architectural design is to optimize the evaluated evolvability and thus create a highly evolvable design, so evaluation approaches are often used in the service of architectural design.

Various approaches have been taken to architecting evolvable software. Chung et al. [51] present a design-pattern-based approach in which they treat adaptability as a quality attribute to be satisficed. Fricke & Schulz [91] present a set of general principles to support changeability in system architectures. ArchWare [176] is a project that takes a model-driven engineering approach, providing languages and a development environment to support engineers in architecting evolvable systems. Focusing particularly on the domain of embedded systems, America et al. [5] present a series of recommendations for architecting evolvable software, including the establishment of a reference architecture and the use of cost-benefit analysis.

This question of how to architect evolvable software is the anticipatory counterpart of the research questions presented in section 1.1—it is preventive, while our approach is corrective. The question of how to architect evolvable software is an important one, but it is useful chiefly for the construction of new systems. This thesis focuses instead on the common problem of evolving, modernizing, or rehabilitating existing systems, which may or may not have been originally engineered with evolvability as an explicit goal.

There are a variety of evaluative approaches as well. Koziolek [118] presents a literature review of work on sustainability evaluation of software architectures. (Koziolek defines the term *sustainability* as encompassing maintainability, modifiability, portability, and evolvability.) Here I highlight a few of the most relevant evaluative approaches.

ALMA (Architecture-Level Modifiability Analysis) [25] is a method for evaluating the modifiability of an architecture in which change scenarios are elicited from stakeholders and the architectural effects of those scenarios are evaluated. Del Rosso & Maccari [70] illustrate a means of assessing the maintainability of an evolving software system. Tarvainen [216] presents an approach for evaluating the adaptability of a software architecture, where *adaptability* is understood to include support for run-time changes "as well as changes in the requirements of stakeholders' objectives." Brcina & Riebisch [36] present an approach for evaluating architectural models for evolvability by modeling traceability links between features, architectural elements, and the implementation. Breivold et al. [37] describe an evolvability analysis method that they developed in the course of a case study on an industrial automation control system. The Evolutionary Scenario Development Method (ESDM) [203] takes a retrospective approach, examining the impact that historical changes have had on architectural quality.

There are also a number of economic approaches to evaluating the evolvability of an architecture. Bengtsson & Bosch [24] attempt to predict the maintenance effort that an architectural design will require by calculating the expected effort for each change scenario. Anwar et al. [7] quantify maintenance cost using a model that incorporates factors such as system novelty, turnover, and documentation quality. Engel & Browning [79] suggest using the economic theory of real options to support architectural adaptability. Bahsoon & Emmerich [10] present a model to predict the stability of a software architecture, also using real-options theory. Cai & Sullivan [41; 42] developed a formal model and tool for reasoning about the technical and economic implications of design modularity, particularly with respect to evolvability.

What all this work on evolvability has in common is that it is concerned specifically with ensuring that the future evolution of a system will be easy, not with planning the details of how such an evolution will be carried out. That is, both areas of work are concerned with *evolvability* as a quality attribute rather than *evolution* as an activity. The work described here, on the other hand, focuses on modeling evolution as an activity; thus, evolvability is relevant here only in the sense that improving evolvability may be a goal of an evolution.

7.4 Planning software architecture evolution

We finally come to the body of research to which this work belongs: models and tools to support software architects in planning evolution. There have arisen a number of different perspectives on the topic of planning architecture evolution, which I summarize under the following subheads:

Capturing architectural transformations. A number of researchers have proposed formal models that can capture structural and behavioral transformation [13; 14; 106; 207; 208; 221]. For example, Wermelinger & Fiadeiro [221] use category theory to describe how transformations can occur in software architecture. Their approach separates computations of a system from its configuration, allowing the introduction of a "dynamic configuration step" that produces a derivation from one architecture to the next. Architecture in this sense is defined by the space of all possible configurations that can result from a certain starting configuration. Fahmy & Holt [81] and Grunske [106] present ways of specifying architectural transformations using graph rewriting systems. Barais et al. [13; 14] present an approach for specifying architectural transformations inspired by aspect-oriented programming. These refactorings are shown to preserve architectural behavior. Spitznagel & Garlan [207, 208] focus on the transformation of architectural connectors as a way to augment the communication paths between components.

While such formal approaches lay a foundation for architecture evolution operations, they differ from the approach described here in that they do not provide specialization for specific classes of transformation and systematic reuse. Moreover, while they can provide some support for characterizing forms of evolution correctness, they do not address issues of evolution quality. Finally, these approaches lack empirical validation. Several of the cited papers [81; 106; 221] contain no empirical validation of any kind. Barais et al. [13] uses a small, artificial example of a banking application. Spitznagel & Garlan [207], in their work on connector transformation, carried out a small case study involving enhancing a Java RMI connector with Kerberos authentication. However, this was a small example in laboratory conditions, not an observation of real-world software engineering practice.

Another area of research worth mentioning here is work on *architectural tactics*, a concept introduced by the Software Engineering Institute as a means of characterizing architectural decisions that are used to achieve a desired quality attribute response [8]. Conceptually, architectural tactics and evolution operators are quite different. An evolution operator represents some real-world operation carried out over time in the course of a software system evolution. Architectural tactics, on the other hand, do not entail any notion of evolution over time; rather, a tactic encapsulates a set of decisions relevant to a quality attribute scenario. However, despite the importance conceptual differences, there are certain similarities between the two ideas; indeed, many architectural tactics can be reinterpreted as operators. For example, an availability tactic based on incorporating redundancy in an architectural design might give rise to an evolution operator that adds a failover component.

Recurring patterns of architecture evolution. In recent years, Tamzalit, Le Goaer, and others have investigated recurring patterns of architecture evolution, primarily with respect to component-based architectures [129–133; 191; 214; 215]. They use the term *evolution style* to denote a pattern for updating a component-based architecture. They provide a formal approach based on a three-tiered conceptual framework.

However, unlike the work described in this dissertation, they do not explicitly characterize or reason about the space of architecture paths, or apply utility-oriented evaluation to selecting appropriate paths. In addition, they seem to have relied exclusively on fictitious examples rather than observing real evolutions. Several of their papers [130; 132; 133] use an example of a client–server architecture based on an example given by Cheng et al. [50]. In a 2007 paper, Le Goaer & Ebraert [129] give a couple of other examples: a banking application evolving to accommodate different account types and a chat application evolving to support two kinds of users.

In a more recent paper, Tamzalit & Mens [214] give yet another example, which they call EShop: an Internet shop application evolving to a client–server architecture. All these examples seem to be artificial; there is no suggestion in the papers that the evolutions described were carried out or that they were based on observation of real evolutions.

A more recently developed style-based approach, building on our work as well as that of Tamzalit, Le Goaer, et al., is a framework developed by Cuesta et al. [63]. Like us, they consider an evolution scenario in terms of evolution steps capturing the architectural evolution of a system. However, they emphasize that the decision-making process in architecture evolution should incorporate other kinds of architectural knowledge beyond just structural information, and they make modeling of architectural decisions a centerpiece of their approach. Their approach is based on local reasoning; evolution decisions are selected in response to specific conditions which can bring them about. Thus, unlike our approach, they do not provide any way of evaluating evolution paths holistically to identify the optimal path, nor any way of making utility-based trade-offs among competing concerns. To illustrate their approach, Cuesta et al. present what they call "a real-world case study," although it appears to be a fictitious example that they contrived based on general understanding of a domain rather than an actual evolution that was carried out, and it is not a formal case study of the sort that I have presented in this dissertation.

Another interesting approach is Yskout et al.'s change patterns [228], which provide a framework for evolving multiple artifacts (in particular, requirements and architecture) in tandem. This work is focused specifically on co-evolution of related artifacts and is not a general theory of software architecture evolution.

Making trade-offs among architecture evolution paths. Another area of work is trade-off analysis for architectural evolution. Within this category, the research that most closely resembles ours is the recent work of Kang & Garlan [116]. Their approach, like ours, is based on modeling candidate evolution paths, which are defined in terms of a sequence of architectural descriptions of a system over time. Also like us, they use cost-benefit analyses to evaluate and select among these evolution paths. However, they view evolvability as an architectural quality that is uniquely important to system evolution; thus, their evaluation framework is built around the idea of distinguishing costs and benefits that are related to evolvability from those that pertain to other qualities.

Also notable is the work of Brown et al. [39], who present an approach to iterative release planning based on analysis and selection of development paths, where each development path consists of a sequence of releases. Their analysis is based on measurement of architectural design dependencies as represented by design structure matrices.

Finally, Ozkaya et al. [177] propose to use techniques from options theory to determine investments in introducing flexibility into a system. In their model, part of the value of applying an architectural pattern is that it affords architects the ability (but not the obligation) to take subsequent design actions; this can be viewed as a real option in economic terms.

None of these methods provide support for specifying architectural transformations, defining evolution constraints, or capturing domain-specific evolution expertise. All three of them focus specifically on analysis of architecture evolution alternatives and do not link this to a broader theory of architecture evolution planning.

Capturing architecture evolution decisions. Zalewski et al. [230] focus on decision-making processes in software architecture evolution. They present a methodology for capturing evolution decisions in a decision map (similar to a conventional mind map). This work focuses solely on capturing and relating decisions and does not directly represent architectural structure. In addition, while our work focuses on providing guidance to help architects make good decisions, Zalewski et al. leave architects to make decisions on their own; their focus is capturing these decisions.

Model-driven reengineering. There has also been work on evolution in the model-driven engineering community. A particularly significant effort in this area is the Object Management Group's Architecture-Driven Modernization initiative, which aimed to develop systematic approaches to model-driven reengineering [169]. This initiative led to the development of the Knowledge Discovery Metamodel [170], which defines an interchange format for reengineering tools. The Knowledge Discovery Metamodel was later standardized as ISO/IEC 19506 [110].

Model-based reengineering has seen some industrial use. Pérez-Castillo et al. [181] present a survey of reengineering tools that have been used industrially, several of which provide some support for model-based restructuring of a software system. Most of these are fairly limited, but they do show how model-based approaches to software evolution can be made industrially relevant.

Van Deursen et al. [218] give a useful survey of the state of the art in model-driven evolution approaches, focusing on limitations and directions for future work.

Case studies in architecture evolution. There have been some exploratory case studies examining instances of architecture evolution and attempting to draw conclusions therefrom. Unlike the work presented in this dissertation, these case studies do not attempt to link their results to a broader theory of architecture evolution; they do, however, provide some useful generalizations and prescriptions.

Erder & Pureur [80] present a real-world case study drawn from their professional experience in the banking industry. They present the case of a loan-servicing company that was migrating from a mainframe system to an event-driven, service-oriented architecture. Based on this case study, they provide advice on how to organize architecture evolution steps into waves and plateaus. The advice is pragmatic in nature, suggesting that introducing major infrastructure changes (waves) should be followed by periods of relative stability to permit adjustment to new infrastructure changes (plateaus).

Antón & Potts [6] present a retrospective case study examining fifty years of requirements evolution in the telephony domain. They observed that the evolution they studied adhered to a pattern of punctuated equilibrium, characterized by large expansions of functionality that were often followed by periods of retrenchment in which the number of services receded slightly.

Bratthall [35] conducted a series of studies examining the impact of architectural understanding on evolution quality. Based on the results of these case studies, Bratthall argues that improving architectural understanding of a system can reduce development time.

Domain-specific work on architecture evolution. Finally, there is work that addresses architecture evolution in the context of a specific architectural style, such as Darwin [144] and C2 [217]. Like the work described in this dissertation, these approaches can take advantage of domain-specific classes of systems and thereby achieve analytic leverage, as well as tool support for evolution. However, these approaches are limited to a particular architectural style.

Outside the research literature, there are plenty of writings that describe, in practical terms, examples of what we would call architecture evolution. A number of examples can be found, for instance, in the IBM Redbooks series, which provides guidance for practitioners on topics such as migrating an Oracle database to DB2 [47] or carrying out a version-to-version WebSphere migration [229]. But again, such sources are aimed at characterizing a single evolution domain and do not relate the example to a general approach to architecture evolution.

7.5 Summary

My research fits comfortably into the final category of work, described in section 7.4: research on models and tools to support software architects. The research areas described in sections 7.1, 7.2, and 7.3—software evolution, software project planning, and software architecture evolvability—are useful for setting the context for this thesis, and to some extent may be seen as complementary to my work (e.g., as noted in section 7.2, project planning and architecture evolution modeling could be used in conjunction), but this thesis quite clearly does not belong to any of those bodies of work.

It does, however, have much in common with the work described in section 7.4. In that section, I mentioned some of the ways in which our approach distinguishes itself from particular other approaches. But it is useful here to summarize the main characteristics that make this thesis research unique. In general terms, the most crucial differences that distinguish our approach from other approaches that have appeared in the literature are:

- Our approach supports path-based reasoning about the space of possible evolutions. Our approach is concerned with defining, analyzing, evaluating, and selecting among multiple candidate paths of evolution leading from the current state of a system to a desired target state. This distinguishes it from approaches that consider only a single possibility for an evolution, or approaches that only model candidate evolution plans without reasoning about them or selecting among them.

- Our approach can be specialized to particular domains of evolution. A key idea in our approach is the use of evolution styles to encapsulate a set of operators and analyses relevant to a domain of evolution. Other approaches tend to be either fully generic (lacking any support for domain specialization) or fully domain-specific (applicable only to a single domain and otherwise useless).
- Our approach has been evaluated empirically. Other work in this area has tended to rely heavily on artificial examples—evolutions imagined in general terms but never actually carried out. Our work is distinguished by the careful and detailed case studies we have conducted to evaluate it empirically (chapters 4 and 5). In addition to this empirical evaluation, we have also undertaken a detailed theoretical study of specification and verification issues (chapter 3) as well as some prototype tool development work to examine issues of implementability (chapter 6). Together, these different evaluation approaches provide a validation that is quite robust in comparison with existing work in this area.

A few other approaches have one of these elements. For example, Brown et al. [39] also do path-based reasoning, Le Goaer et al. [132] also focus on domain specialization, and the work of Bratthall [35] is heavily empirical. But ours is the only work that provides a general approach to aid software architects in planning evolution based on principled consideration of evolution alternatives, leverages domain expertise to support specialized analyses, and has been subjected to rigorous empirical evaluation.

8 Conclusion

In this conclusion, I reflect on the significance of this thesis and its limitations. Section 8.1 discusses the thesis contributions. Section 8.2 points out limitations of the work. Finally, section 8.3 suggests directions for future work.

8.1 Contributions

The principal contribution of this thesis is an approach for reasoning about software evolution at an architectural level. This main contribution incorporates several parts:

- A theoretical framework defining a set of concepts and abstractions to facilitate architectural reasoning about software evolution
- A means of defining operators that characterize the architecturally significant evolution operations that can be applied to a software system, including their preconditions, structural effects, and analytical properties
- A means of defining constraints that specify what constitutes a permissible path of evolution, in a way that is amenable to automated verification
- A means of defining analyses that allow evaluation of the quality or utility of a candidate evolution path
- A means of capturing knowledge about specific domains of software architecture evolution in a reusable form

The approach presented in this thesis is the first systematic method for modeling and analyzing multiple candidate evolution plans at an architectural level.

In addition to this primary contribution, this thesis makes several ancillary contributions that are also significant:

- Novel theoretical results on the computational complexity of model-checking a version of linear temporal logic with the addition of rigid variables (also called freeze quantifiers or reference pointers). These results are of interest in this work on software architecture evolution because they characterize the difficulty of evolution path constraint verification. But they are also of broader theoretical interest. As section 3.3 documented, logics very similar to our path constraint logic have been applied to problems in several domains, such as object-oriented data modeling and real-time systems. Resolving the important question of the complexity of the finite-path model-checking problem [147] for this class of logics is thus likely to be of use to researchers in other fields, outside of software architecture.

- Empirical results on the applicability of this research on software architecture evolution, based on two case studies of real-world software organizations. By applying our approach to architecture evolution modeling in two large software organizations—JPL and Costco—I demonstrated that our approach can capture the concerns that arise in practice during major evolution projects. The Costco case study provides particularly robust results in this respect, due to its careful methodological design and rigorous execution, and also because it evaluated all the elements of our approach (in contrast to the JPL case study, which focused on the construction of an evolution model and did not examine the issue of model analysis in depth). But the JPL case study is also important for the applicability argument, because it helps to demonstrate that the approach is applicable to different kinds of organizations—that Costco was not a unique case. By assembling the findings of two case studies in this way, we can make much stronger claims about the general applicability of the research, even if one of the case studies was less formal than the other. The argument is particularly strong because Costco and JPL are extremely different organizations—a retailer and a government-funded laboratory. If our approach can be applied in these very disparate contexts, we can reasonable conclude that it likely has a fairly broad scope of applicability.

  In addition to the results on the applicability of our modeling approach, the Costco case study also produced descriptive results about the state of architecture evolution practice in a real-world software organization. These results help us to position our research, but they may also be of broader use beyond this work. There has been very little work directed at systematically examining the impetuses of architecture evolution, the architecture evolution challenges perceived by real-world architects, and the approaches by which architects manage evolution today. This case study provides some useful results on these questions, beyond its immediate relevance to our research.

- A research methodology for extracting information from architectural documentation in a systematic, replicable way. In addition to its overt contributions to software architecture evolution research, the Costco case study also makes a significant methodological contribution. The use of content analysis to mine architectural elements from architectural documentation is a novel approach that I hope will be of use to other software architecture researchers. Usually, software architecture case studies are fairly informal. In particular, researchers constructing architectural models of systems tend to rely heavily on their own understanding and intuition about the system being modeled. The use of content analyses systematizes the process of model construction, reducing the risk that researcher bias will taint the results, and significantly improving replicability. The Costco case study demonstrated how content analysis can be effectively applied to architectural documentation to achieve these results, and in particular it shows how a coding frame can be designed to extract architectural model elements from content (which may include diverse forms of data such as interview transcripts, textual architectural documentation, and architectural diagrams). It is my hope that this novel methodological approach to empirical software architecture research, or variants of it, might be adopted by other software architecture researchers working with heterogeneous real-world data.

- Findings on the implementability of the approach, based on the development of several prototype tools. These prototype tools have demonstrated how the approach presented in this dissertation can be implemented in practical tools that leverage existing architecture modeling frameworks. In addition to these general results on implementability, they have allowed us to explore how various aspects of our approach can be automated, such as the application of evolution operators and the generation of evolution paths. At the same time, this tool work has revealed areas where significant research challenges still remain, such as relating multiple architectural views of an evolution state and using constraints and evaluation functions to automatically generate evolution paths.

8.2 Limitations

Although some important results about our approach have been demonstrated, it is important to understand the limitations of this work. In this section, I highlight a number of noteworthy limitations.

The approach carries a significant specification burden. The approach presented in this thesis requires significant formal specification, including specification of evolution elements such as operators, constraints, and evaluation functions as well as specification of an evolution space in terms of intermediate architectural states and evolutionary transitions among them. The significant specification effort required may make our approach seem a bit heavyweight, limiting its adoptability.

There are a couple of ways in which this burden is mitigated. First, there is the idea of evolution styles. An evolution style allows a set of operators, constraints, and evaluation functions relevant to an evolution domain to be bundled together in such a way that they can be applied repeatedly to evolution projects within that domain. In this way, the cost of specifying the evolution elements is amortized over a large number of systems. This significantly reduces the specification burden. It also supports a work flow in which evolution elements are *defined* by a small group of people (domain experts with some knowledge of formal methods—perhaps researchers or senior architects, for example) and then *used* by a much larger community (software architects working within the domain). In other words, our approach does not presume that architects using it must have the necessary knowledge and expertise to specify evolution styles themselves; rather, they can take advantage of evolution styles created to support such use.

But even assuming that an evolution style is available to an architect, the architect still has to define the evolution space, including the initial architecture, the target architecture, the intermediate architectures, the evolution transitions (and their decomposition into operators), and the candidate evolution paths. In our initial prototype tools, very little automation was available to support the definition of the evolution space. In subsequent work, we have demonstrated that much of this can be automated, at least in principle. The architect will still need to define the initial and target states, but if the evolution style provides an adequate palette of operators and specifies adequate constraints and evaluation functions to define what constitute a legal and high-quality evolution path, then an automated planner can generate evolution paths by composing these operators automatically, as explained in section 6.3. However, there are still unanswered questions about how this could be integrated with an architecture evolution tool in practice.

Future work might be able to reduce the specification burden further, including the burden of defining evolution styles. For example, instead of requiring operators to be specified in some kind of code, it might be possible to support graphical specification of operators, or to infer operators from examples of architectural transformation. Similarly, constraints must currently be specified in temporal logic, which may seem a bit arcane to practicing architects. But in the future, an intelligent user interface might allow architects to build constraints from a library of constraint templates capturing common classes of constraints such as ordering constraints, timing constraints, and integration constraints.

The approach requires adequate tool support in order to be adopted for practical use, but such tool support does not yet exist. The approach for architecture evolution modeling I have presented in this thesis is inherently dependent on tool support in order to be practical. Without good tools to provide support for constructing and analyzing evolution models, the framework we have defined is of little use. To show that the approach is actually implementable and to examine some of the issues involved in tool development, we developed prototype tools that support key elements of our approach.

But while demonstrating the implementability of the approach in principle is one goal of this thesis work, actually producing a mature tool that could be readily adopted by practitioners is not. None of the prototype tools we have developed are sufficiently complete, usable, or mature that they could actually be adopted for use in a real, industrial-scale evolution project. This is not a limitation of our approach in principle, but it is a limitation of this thesis. Further tool development would be necessary to make the approach adoptable by practitioners.

The empirical work justifying the applicability of this approach consists of just two case studies. I have argued that the two case studies presented in this dissertation provide strong evidence of the applicability of our approach. I have given particularly careful attention to issues of reliability and validity so as to strengthen this claim. In addition, the use of two case studies conducted in very different organizational contexts helps support a claim of generalizability.

Even so, case studies have their limitations, and in attempting to generalize from the results of investigations of just two organizations, we can go only so far. Future empirical work on this kind of approach to software architecture evolution would be of great help in evaluating the scope of our results. A survey-based study (examining a statistical sample of many software organizations) or a multicase study (examining a number of different software organizations in some depth) might be particularly useful in this respect.

There may be some kinds of software organizations for which our approach is less suitable. The two case studies featured in this dissertation were conducted at two very different software organizations. But they were similar in at least one important respect: both organizations featured an architecture group of significant size and maturity. Not all software organizations have an architecture group; indeed, not all software organizations have software architects. The relevance of our approach to organizations where architecture is not formally practiced is unclear.

This is an architecture-level approach, and the fidelity of the implementation to the architectural design is in no way guaranteed. We believe that an architectural approach provides a number of benefits not available to code-level approaches. Reasoning at an architectural level allows us to model high-level concerns such as integration issues and evolution stages. Taking an architectural perspective also grants our approach a kind of generality that is usually not available to code-level approaches, which are frequently programming-language-specific. Finally, an architectural approach is highly scalable—just as applicable to vast, complex, multifaceted systems as to small ones. Code-level approaches tend to operate at a more limited scale.

However, an architectural approach also carries with it certain limitations. One important limitation is that an architectural model is not necessarily tied to the reality of the software system. That is, there can be problems of *architecture conformance*. There is no mechanism that ensures our evolution model is a faithful representation of the system it purports to describe. There is a significant body of research on architecture conformance [2; 28; 162; 200], some of which could be applied here. But I think there is also a need for more research specifically on *evolutionary* architecture conformance—ensuring not only that a system is in conformance with its supposed architecture, but also that conformance is retained as we carry out evolution operations.

8.3 Future work

Throughout this dissertation, I have intermittently noted areas in which some future work would be helpful for clarifying certain issues or advancing the state of knowledge of certain topics. In this section, I focus on a few major future areas of work where I believe a number of significant research challenges remain. In my estimation, each of the following veins is sufficiently rich to be potentially worthy of a PhD thesis itself!

8.3.1 Elaborating the idea of evolution styles

Section 2.6 presented evolution styles as a means of encapsulating knowledge about particular domains of evolution. Formally, an evolution style is simply a collection of operators, constraints, and evaluation functions specialized to some domain.

Conceptually, evolution styles are quite important to our approach. There are a few reasons for this. First, they unify the various evolution elements, relating operators and constraints and evaluation functions. Second, they permit our approach to be specialized to particular domains, which we view as one of the key strengths of the approach. Finally, they help to justify the adoptability of the approach by reducing the burden of specifying evolution elements, since evolution styles provide a systematic way for evolution elements to be defined once but used many times.

However, evolution styles have not figured prominently in our empirical work or our tooling work to date. In our case studies and prototype development, we have examined the applicability and implementability of the individual elements of our approach—operators, constraints, and evaluation functions—but we have not specifically examined the usefulness of the evolution style concept for capturing domains of evolution.

Aside from the need for further empirical and tooling work, however, there are more fundamental questions about evolution styles that need to be answered—questions which we have skirted by defining an evolution style as simply a collection of operators, constraints, and evaluation functions.

How can we make use of multiple evolution styles in a single project? In our work so far, we have generally assumed that an architecture evolution problem will make use of a single evolution style—some evolution style that usefully captures the problem domain. However, there are many good reasons one might want to combine multiple evolution styles. Consider a situation in which there are two evolution styles that are relevant to an evolution scenario. For example, if an evolution incorporates a client-server system evolving to a decentralized peer-to-peer model, as well as a SQL database being migrated to a cloud storage system such as Amazon S3 (so that any of the peers can access the data store), we might want to leverage two evolution styles: one for evolving a client-server system to a peer-to-peer system, and one for data migration. Scenarios analogous to this one arise often in heterogeneous systems in which different subsystems have different architectural styles.

Another possibility is the definition of *mix-in* evolution styles that are designed specifically to supplement conventional evolution styles—for example, mix-in styles to support specialized analyses such as performance analysis. Such a mix-in style could used in conjunction with a conventional style such as a data migration style; the data migration style would define the basic evolution elements (operators for migrating data, constraints on how these operators may be applied, and so on), and the mix-in style could contribute some constraints and evaluation functions for performance analysis. Supporting this sort of style composition intelligently is harder than it seems. For example, a mix-in style for performance analysis would not only need to define new evaluation functions and constraints pertaining to performance; it would also likely need to somehow modify the operators specified in the base style to supplement them with properties needed to analyze performance. It might also need to enrich the architectural style used to define intermediate states, augmenting architectural element types with properties such as latency and throughput. Defining a system for composing evolution styles ad hoc in this way poses formidable research challenges.

How can evolution styles be related? How can one evolution style build upon another? We have thus far assumed that each evolution style is designed from scratch, in isolation. But it is reasonable to want to define a new evolution style by building on an existing one—or more specifically, by specializing an existing one. Such specializations might be needed for technical reasons (e.g., specializing a very general data migration style with a style specifically for fine-grained modeling of SQL database migrations) or for organizational reasons (e.g., a certain company wants to specialize an evolution style by adding its own operators, constraints, and evaluation functions specific to how that company does business).

Although this is a sensible idea, there are many questions about how to implement it. How should the evolution elements in a substyle relate to the evolution elements in a base style? What should be the inheritance model? Should we support multiple inheritance? Would it be better to have some compositional model rather than an inheritance-based design? Can a substyle delete or modify elements in a base style, or merely supplement them?

How can we ensure that evolution styles are defined at an appropriate level of generality—specific enough to be practically useful, general enough to be broadly reusable? It is very easy to define an evolution style so high-level and nebulous that it is of very little practical benefit.

A partial solution to this dilemma is the one mentioned above: an inheritance model for evolution styles. With inheritance, we could define a family of related evolution styles, with more specific styles inheriting from more general ones. But this doesn't entirely solve the problem. Even a root evolution style that serves as a base for many more specific styles has to be carefully designed so as not to be uselessly general, and even the styles at the leaves of an inheritance tree need to be general enough that they can be reused, or they aren't worth defining at all. Moreover, designing such a hierarchy of styles requires good judgment about what to include and (perhaps more importantly) what not to include; an overly fine-grained inheritance hierarchy can create more problems than it solves.

How should a library of evolution styles be maintained and used? Evolution styles may indeed be effective at encapsulating reusable knowledge about domains of evolution—presuming an architect has access to appropriate evolution styles in the first place. There are a number of research challenges pertaining to how evolution styles should be stored in such a way that they can be fruitfully discovered and applied by architects. One might imagine software companies maintaining evolution style repositories, or global evolution style repositories available online. But what metadata do we need to have to make such repositories easily searchable? By what kind of taxonomy can we meaningfully categorize evolution styles? How can duplication and other kinds of redundancy be avoided? Can we devise ratings systems to help architects distinguish between good and bad styles?

How does an evolution style relate the evolution elements within it? An evolution style is not just a haphazard assortment of operators, constraints, and evaluation functions; rather, it is a family of closely related elements that have been developed to describe a particular domain. The elements in a typical evolution style will therefore have a lot in common. They will refer to the same architectural properties and architectural element types, the constraints will govern the use of the style's operators, and so on.

Currently, however, these relationships are not enforced in any way, nor even well understood. Ideally, we would like some way to guarantee the consistency of the style—to make sure that the architectural properties that an evaluation function is reasoning about are the same ones that are being set by the operators. But it is not clear how this should happen.

8.3.2 Developing sophisticated, mature architecture evolution tools

The tooling work presented in this dissertation consists solely of preliminary prototypes. While these serve as useful proofs of concept and help to demonstrate the implementability of the approach, they are not nearly mature or complete enough to be used by practitioners in their present state. A great deal of tool work remains to be done, and a number of significant challenges remain as well. Of course, some of this is just a matter of engineering effort—putting in the work to realize the vision we have laid out. But there are also significant research questions that have not yet been answered with respect to tool development. Among the most significant questions that would benefit from future research are:

How can various aspects of architecture evolution modeling and analysis be automated? Automation is very important for developing practical architecture evolution tools. In our previous work, we have demonstrated a couple of ways that automation can be used to ease the burden of architecture evolution modeling, including automation of operator application (see sections 4.3.3 and 4.4.3) and automation of evolution path generation (see section 6.3). But there is more that can be done to improve these methods, and there are other areas where automation would also be helpful. For example, a very useful feature would be semiautomated creation of evolution operators, in which the user defines the structural transformations that an operator entails by example, and a tool produces a formal operator definition.

How can an evolution tool effectively leverage existing frameworks for architecture modeling? In the MagicDraw tool work, we discovered a number of challenges in implementing our approach as a plug-in to an existing architecture modeling framework. These challenges will need to be resolved to make a full implementation practical.

How can we support an evolution graph definition work flow that makes sense to the user? An evolution graph is a simple thing in principle, but actually defining one is a somewhat intricate process. Transitions leading to intermediate states are defined in terms of a series of operators, each with its own structural effects and preconditions. For this model to be accessible to the user, it has to be presented in a sensible way. One possibility is a transition specification mode, in which the user can see the effects of operators as they are applied and composed together to form a transition. However, in our MagicDraw tool work, we found that such a feature was very difficult to implement in practice due to the limitations of the modeling tool API.

How can we best maintain the consistency of the evolution model? In a complete evolution model in which all evolutionary transitions are defined in terms of the operators that compose them, the transitional states are overspecified. For an intermediate state S, there can generally be multiple evolution paths leading from the initial state to S. Each of these evolution paths can be understood structurally as the composition of the operators that make up its transitions. If the structural transformations entailed by one path produce a different architectural state than those entailed by another path, there is a model inconsistency.

In theory, this does not present a serious problem. We simply declare by fiat that an evolution graph must be consistent. But enforcing this in a tool poses challenges. If the evolution graph is defined by hand, how do we get the user to define the paths in such a way that consistency is guaranteed? If an inconsistency does arise, how do we report it in a way that makes sense to the user? If the evolution graph is defined automatically by a planner, how do we ensure the consistency of the generated paths? If the user wishes to modify a generated path, how do we reconcile the other paths to achieve consistency? Although the issue of model consistency seems conceptually simple, there are daunting research challenges just below the surface.

8.3.3 Evaluating and enhancing the usability of the approach

One important criterion that I have not provided any formal validation for is usability. In both the case studies, I was the one doing all the modeling; and none of our prototype tools were ever tested with real users. As a result, there are many outstanding questions about how easy it is for practitioners to use the approach, and how the approach may be made more easily usable. Among the most interesting such questions are:

How easy or difficult is it to specify operators, constraints, and evaluation functions? This is perhaps the most obvious usability question to arise from our approach. Can practitioners actually specify these evolution elements in the way we have prescribed, or are our specification languages too arcane for practical use? (We justify the difficulty of specifying evolution elements by noting that only evolution style designers need to do the specification work, and then many architects can use them. But still, *someone* needs to do the specifying, so it had better be doable with a modest amount of training.) This is a rather straightforward question to answer; it could be addressed through ordinary usability testing.

How can we display an evolution graph in a way that facilitates understanding and easy access to information? In an industrial-scale evolution project, an evolution graph would contain a lot of data—potentially dozens of states, each with perhaps hundreds of architectural elements, in addition to the transitions among the states, each made of a few operators. How can we display an evolution graph in a way that makes sense to the user? Displaying a network of undifferentiated nodes and edges (as we did in the JPL case study) is quite uninformative; users will easily lose track of which nodes are which. A clearer approach is to display miniature representations of each state within the corresponding node, as in the notional illustration in figure 2, but this becomes impractical for large and complex architectures. It is even less clear how best to represent an evolution transition in terms of its operators in such a way that the user can easily understand and manipulate those operators.

How can we make operators, constraints, and evaluation functions understandable by users? We have argued that specifying evolution elements need not be exceptionally easy, because ordinary users need not specify them; evolution style designers (senior engineers or researchers) can specify them, and they can be reused across many projects. But even so, users still need to be able to *understand* the elements they're using. Indeed, the problem of understanding is all the more acute when the person using the evolution style is not the person who designed it. How can we represent evolution elements to users in an understandable way? Consider operators, for example. An operator's name may be fairly opaque; it's not immediately clear what the effects of a *wrapSystem* operator might be. Its formal definition is even more opaque; an ordinary user shouldn't have to dig into the formal definition of an operator in order to figure out what it does. How can we present information about the operator to users in a way that helps them to understand its effects and its preconditions? For example, when the user selects an operator from the palette, perhaps we could generate some kind of before-and-after illustration showing what the operator's effects are on a trivial example. But there are significant technical challenges in generating such a visualization. Similar questions apply to constraints and evaluation functions.

How can we localize constraint violations so that the user knows what to fix? Formally, constraints are simply predicates over evolution paths; for a given path, a constraint says, "Yes, the path is valid," or "No, the path is not valid." But a simple yes-or-no answer is not particularly helpful. Troubleshooting why a constraint fails for a path is likely to be a very frustrating process unless we can do some kind of constraint violation localization to track down where the problem is. Otherwise, the fault could be literally anywhere: a single property of a single port within a single state, for example. However, there are significant theoretical challenges to doing this kind of localization.

How can we provide a user interface that facilitates flexible reasoning about trade-offs? A key purpose of evolution styles is to facilitate trade-off-based reasoning. This can be accomplished through the definition of a utility evaluation function as a weighted composite of other functions that evaluate more primitive qualities such as cost, duration, availability, and so on. But if this utility function is hard-coded in an evolution style, how can we give style users the flexibility to explore the trade-off space by adjusting these weights? What features can we provide to facilitate easy and informative exploration of the trade-off space, and how can these features be implemented with respect to the evaluation functions defined in a style?

Better yet, is it possible to automate the trade-off analysis, and to present the user with a meaningful summary of the trade-offs among paths? Can we automatically characterize the most significant distinguishing factors among the paths? For example, such an automated trade-off analysis might highlight that the selected path has an estimated completion time of 29 months and an estimated failure risk of 8%, while an alternative path has a much faster completion time but a significantly higher risk of failure. Automatically identifying these trade-offs and presenting them in a helpful way presents a number of challenges.

8.3.4 Modeling and relating multiple views of a system

In reasoning about a software system, it is often helpful to consider multiple architectural views of the system [53]. For example, in an evolution involving the migration and redeployment of software components, it would likely be helpful to consider both a component-and-connector view and a deployment view of the system. In section 2.1, I described how our approach can accommodate representation of multiple views of each evolution state, and in a paper [19] we showed an example in which a component-and-connector view was represented as a UML component diagram and a deployment view was represented as a UML deployment diagram. However, a number of research challenges remain.

How can relationships across views and across models be represented? It is not enough merely to have multiple views of each evolution state; we also want these views to be related. For example, it isn't enough to have a component-and-connector view representing the run-time structure of a system and also a deployment view representing its physical structure; we also need to relate those views to indicate which components are deployed on which hosts. If our views are different UML diagram types, as in our exploratory work on relating views of evolution states [19], this is fairly straightforward; UML diagram types can be related because they are merely different projections of a shared model. But relating different types of models—say, an Acme component-and-connector view and a UML class diagram representing a module view—is much harder. How can an Acme model incorporate references to a UML diagram, or vice versa? And how can the integrity of such references be enforced and preserved as the models change?

How can other system views besides conventional architectural views be used to inform evolution planning? We need not limit ourselves to considering conventional architectural views of a system. By including other, nonarchitectural system views, we can gain significant analytical leverage. For example, given the importance of human factors in system evolution, it might be useful to have an *organizational* view alongside the architectural views. Such a view could contain information on development teams and their respective competencies and specialties. With such a view defined, we could define constraints restricting which teams can work on which parts of the system and which skills are required for which tasks, operators that effect organizational transformations such as restructuring or training a development team, and evaluation functions that make use of information in the organizational view to estimate concerns such as cost and effort in a more precise way. But although this idea is appealing in principle, it is not clear what guiding principles should govern the introduction of nonarchitectural views, or how such views might be defined or related to other views.

How can constraints across views be enforced? One of the appeals of modeling multiple views is that we can write constraints to reason about the multiple views and the relations among them—for example, a constraint that a component of a certain type can't be migrated to a particular host environment until some necessary configuration has been done. But what modifications must be made to our constraint specification language to accommodate such constraints? Currently, the constraint specification language relies on the architectural modeling language to provide a way of specifying architectural predicates. But what happens when there are multiple architectural modeling languages, each with its own way of specifying architectural predicates?

How can tools be developed that support multiple views? In addition to the theoretical challenges discussed above, implementing full support for multiple views would also present practical implementation challenges. In section 6.2, I discussed how architecture evolution tools can be implemented as plug-ins to existing architecture modeling tools in order to leverage existing support for definition of architectural models. But few architecture modeling tools provide good support for modeling and relating multiple views of a system. Most UML tools support models with multiple diagram types, but there are not mature tools for relating different types of architectural models or for relating architectural models to other kinds of system model. Thus, the introduction of support for multiple views significantly complicates implementation.

8.3.5 Making the approach scalable to systems of systems

Our approach as it is described in this dissertation presumes a simple and sequential state of affairs. An architect defines a single evolution graph in which each state contains a complete representation of the system. Operators are modeled as being carried out in sequence; an evolution transition comprises a sequence of operators that are applied in turn.

This model may not scale very well. In the evolution of a very large system, there may be many architects and engineers, each with responsibility for some small portion of the overall architecture. In addition, many development teams may be working simultaneously on different portions of the system; that is, development operations are carried out in parallel rather than in sequence. Extending our approach to scale to such scenarios poses substantial challenges.

How can an architecture evolution tool be responsive to the structure of a software organization, allowing different stakeholders to focus on different portions of a system? Large evolution efforts require different teams to focus on different portions of the system. Often, different architects will plan different aspects of the evolution. How can an evolution tool support this pattern of work, allowing architects to collaborate in planning a major evolution? One might imagine some kind of distributed system in which different architects work on a shared evolution model, with some system of permissions to determine which architects may work on which portions of the system, but there are numerous questions about how this idea could be realized.

How can we extend our approach to accommodate parallel execution of evolution operators? The assumption of sequential application of operators is ingrained in our model. The evolution graph is defined in terms of transitions comprising sequences of operators. Constraints are defined in a logic that presumes a linear evolution path. And operators themselves are defined in a language that assumes that an operator has perfect knowledge about, and exclusive control over, the state of the entire system for the duration of its application. To accommodate definition of operators occurring in parallel, we would need to throw out all these assumptions, revisiting each element of our modeling approach. An evolution path might be conceived as comprising multiple parallel threads of effort. The operator specification language would need to be reinvented to accommodate the fact that multiple operators may be occurring in parallel. This presents a number of difficulties. For example, we now must consider the possibility of conflicts among operators occurring in parallel (e.g., two operators modifying the same connector at the same time). And the path constraint specification language would need to be redefined as well. If an evolution path is a set of parallel threads rather than a linear sequence, then linear temporal logic may no longer be a good basis for our path constraint language.

Can the idea of planned architecture evolution be extended to sociotechnical ecosystems that are not under the control of a single organization? Sociotechnical ecosystems have become an increasingly hot topic in software research. How can the architecture of a sociotechnical ecosystem be changed in a principled manner? Evolving a system when there is no single entity that has control over that system is a challenging prospect. But effecting major changes to such a sociotechnical ecosystem is necessary from time to time, and organizations that manage such ecosystems might benefit from tools to help them plan evolution. How can such evolution be planned in the face of radical uncertainty about the actions of other ecosystem participants?

8.3.6 Resolving unanswered questions on automated planning

Section 6.3 reviewed our work on automated planning, demonstrating how evolution paths can be generated automatically based on existing automated-planning technologies. While this work is a useful first step, there are many significant research questions that remain before such automation can be made practical.

How can we best generate multiple candidate paths and present them to the user? In the work in section 6.3, we delegated to the planner not only the task of generating candidate paths, but also that of selecting an optimal path. However, it might be desirable to keep the architect in the loop. Rather than present a single, supposedly optimal path to the architect, it might be better to present multiple candidate paths, allowing the architect's experience and judgment to play a role in path selection. However, it is not clear how we might best generate multiple paths. One option might be to run the planner multiple times, each time optimizing a different metric.

How can we model transitions consisting of multiple evolution operators? Evolution operators capture fairly fine-grained architectural changes: installing an adapter, migrating a database, upgrading a message bus, and so on. In conceptualizing an evolution path, it is helpful to think of evolution transitions as being somewhat coarser, with each transition potentially comprising multiple operators. This streamlines the evolution graph, making it more comprehensible to architects and simplifying analysis. In our automated-planning work, however, we treated evolution operators as synonymous with the evolution graph transitions. It would be better for a planner to aggregate operators into larger transitions by identifying particularly significant points within the evolution to serve as the nodes of the evolution path. However, it is not clear on what basis it should select these significant points.

How can the translation of architecture evolution problems into PDDL specifications be automated? In the work in section 6.3, an evolution scenario was encode in PDDL manually. That is, although the generation of the evolution paths was automated, the definition of the scenario in PDDL was not. For this research on automated planning to be practical, we must be able to translate an evolution scenario into a planning problem automatically. This, however, entails significant challenges. How can an operator specification such as the one shown in figure 1 on page 15 be automatically converted into a PDDL action? How can a constraint written in our extended temporal logic be realized in PDDL? And perhaps most difficult, how can an evaluation function, defined in a general-purpose programming language, be translated into a PDDL metric? It may not be possible to solve these problems in the general case; we may need to impose restrictions on the form that evolution elements can take if they are to be translated into PDDL. But even so, automating the translation of an architecture evolution scenario into a planning problem would remain a formidable research challenge.

How can evolution styles be incorporated into an automated-planning approach? One concept that was not examined in the work in section 6.3 is evolution styles. If we take an automated-planning approach, can we use the idea of evolution styles to facilitate reuse of portions of planning specifications within domains of evolution? The obvious idea is to equate PDDL domain files with evolution styles and PDDL problem files with individual evolution scenarios. This correspondence doesn't quite work out exactly, however, since domain files may sometimes contain information that is scenario-specific, and problem files often contain information corresponding to elements of an evolution style (such as the goal metric, which corresponds to an evaluation function).

What other approaches, aside from automated planning, could be applied to automated evolution path generation? We have focused on automated planning using PDDL as a method for automatically generating evolution paths. This was a natural choice, given the straightforward correspondence between the architecture evolution problem and the planning problem. But it is only one possible approach. Other approaches not yet explored, such as constraint satisfaction or techniques from operations research, might be fruitful avenues for exploration.

8.3.7 Developing a general operator specification language

The operator specification language presented in section 2.2 is simple but effective for specifying architecture evolution operators. But is there anything fundamentally different about specifying architecture evolution operators, as opposed to other kinds of architectural transformation? There are a variety of subfields of software architecture research where specifying operators is useful: run-time adaptation, architecture differencing and merging, and model-driven architecture, for example. Indeed, many architectural-model transformation languages have been developed, from languages like QVT for describing transformations in UML and its kin [171] to languages like Stitch in the context of run-time adaptation [49]. But there has been little work on unifying or distinguishing these different approaches to architecture transformation. Specifically, more work is needed to address the following questions:

What needs do different software architecture communities (including both researchers and practitioners) have for model transformation languages? Although the need to formalize architectural transformation arises in multiple domains, it is not clear to what extent these different domains' requirements overlap. Can we define an operator specification language that is general enough to be used for architecture evolution, run-time adaptation, model-driven architecture, and so on? Or are there fundamental reasons that different fields of research and practice require their own formalisms?

What approaches have been developed for formalizing transformations of architectural models? There have been various surveys of model transformation approaches over the years [66; 155; 156; 201], but these have not devoted attention specifically to the issue of general-purpose, architecture-level languages for model transformation. Much of the existing work on model transformation is not architectural at all, and does not leverage the structural properties that software architectures have. That work which does deal with architectural models is often concerned with translating architectural models to or from something else, such as lower-level models or source code, not with expressing transformations that can be applied to a software architecture. Finally, many existing model transformation approaches are quite arcane and mathematical, not intended to be accessible to software practitioners. The question of what approaches exist to support representation of software architecture transformations in general is quite different from the question that these existing reviews have examined.

How can we define a general-purpose language for specifying architectural operators? Based on the answers to the first two questions in this list, it should be possible to define a set of requirements for such a specification language and to design a language that meets a wider range of needs than the ad hoc operator specification language described in section 2.2.

8.3.8 Analyzing technical debt

Technical debt is a metaphor for software development and maintenance introduced by Cunningham [64] in 1992:

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation [...].

Researchers have recently been devoting increasing attention to technical debt [123], particularly its application to software architecture.

Technical debt is often thought of as something negative, something to be avoided—a way of understanding the chaotic state into which some systems fall. In this sense, technical debt can be understood as a cause of evolution. That is, sometimes the purpose of an evolution effort isn't to add features or improve system quality; sometimes evolution is necessary just to correct a state of disarray that has arisen through years of neglect—to pay off accrued technical debt by refactoring the system architecture.

But technical debt need not be a bad thing. Sometimes it makes sense to go into debt. When time is of the essence, favoring expediency at the expense of architectural quality may be exactly the right choice—provided that the technical debt is promptly repaid. This suggests the possibility of evaluation functions to analyze technical debt. An architect could model two potential paths—one in which a feature is implemented slowly and carefully, and another in which the feature is implemented as expeditiously as possible and problems of architectural quality are addressed later—and explore trade-offs between them. This idea raises a number of questions.

How can we specify evaluation functions to analyze technical debt? Technical debt is often referenced as a metaphor, but it is seldom quantified and formally evaluated. How can we reason about technical debt in a meaningful way using evaluation functions? Can recent research on metrics for architectural debt [167] be adapted to our approach? Can we define technical debt in terms of the properties and architectural structures present in an evolution path?

Can evaluation functions for technical debt be reused across multiple instances of evolution? Analyzing technical debt in the context of a single system might be doable, but can we define more-general technical debt analyses that can be reused across systems (in the same sense that we can define a general cost analysis or a general availability analysis)? What are the right abstractions and models for reasoning about technical debt in the context of architecture evolution? Can we develop a general, broadly applicable theory of technical debt? Can we create an evolution style for technical debt analysis?

How can technical debt be managed in the presence of uncertainty? How can analyses of technical debt accommodate a lack of certainty with respect to the impact of delaying necessary maintenance or to the benefits of speedy release?

8.3.9 Incorporating uncertainty into the model

The model described in chapter 2 is entirely deterministic. The initial and target states are known with perfect certainty, as are the effects of operators, and it is assumed that an architect will be able to carry out an evolution path without deviation or misadventure. In reality, of course, there are all kinds of uncertainties and risks in any major evolution effort. The initial architecture may be uncertain; operators may have unexpected effects; unforeseen contingencies may arise. To take a concrete example, an evolution effort may have a dependency on a system with which it is supposed to integrate, and the date on which that system is to be delivered is uncertain.

We have recently done a bit of exploratory work on the topic of modeling uncertainty in architecture evolution.[20] Thus, in this section, rather than ask open-ended questions as in the other sections on future work, I present some early concrete ideas about how uncertainty could be incorporated into our model of architecture evolution.

As noted above, there are many different forms and causes of uncertainty in architecture evolution. But from a modeling perspective, we can abstractly think of them all as instances of something more general: modifications to the architectural model that may occur nondeterministically at certain points in an evolution. By thinking about the uncertainty problem in this very abstract way, we can gain significant generality (in terms of the kinds of uncertainty we can address).

With this in mind, I propose that we add uncertainty to our model through the introduction of *pseudo-operators* that may occur nondeterministically at various points in an evolution path. Just like regular operators, pseudo-operators are defined by the evolution style designer and comprise the same basic parts: a definition of their preconditions, a specification of their architectural effects, and some analytical properties. Unlike operators, however, which represent tasks for an engineering team to carry out, pseudo-operators occur nondeterministically. Whenever the preconditions of a pseudo-operator are met, that pseudo-operator occurs randomly with some probability that is defined as part of the pseudo-operator specification. Pseudo-operators can be conceived as operators that are applied by the environment and are beyond the control of the people carrying out the evolution.

The introduction of pseudo-operators complicates what it means to find a solution to an architecture evolution problem, and in particular it complicates what we mean

[20] I would like to acknowledge here the work of Fu Maosong, a master's student who did an independent-study project with us, exploring how the ideas in this section could be implemented using existing formal modeling tools.

by “evolution plan.” Now an evolution plan cannot just be an evolution path—a sequence of operators leading from the initial architecture to the target architecture—because we can’t count on which pseudo-operators will occur. Instead, an evolution plan is a strategy or policy that defines, for any possible evolution state, what operator we should apply next to optimize our expected outcome.

Observe the similarities to Markov decision processes, which also have as their solution a *policy* that determines, for any state, which action a decision maker should select. I suspect that a nondeterministic architecture evolution problem as I have defined it above may be modeled as a Markov decision process in a fairly straightforward way—provided that we choose the state space for the Markov decision process carefully.

In principle, this approach should be flexible enough to capture multiple kinds of uncertainty pertaining to architecture evolution, including:

- Uncertainty about the initial architecture. In this case, we can introduce an architecture recovery operation that transforms the architectural model.
- Uncertainty about operator effects. In this case, an operator may be followed by a pseudo-operator that transforms the architectural model.
- Uncertainty about operator qualities. In this case, an operator may be followed by a pseudo-operator that modifies some system property representing a quality such as time or cost.
- Uncertainty about exogenous events that may occur (such as integration delays in the example at the beginning of this subsection). In this case, a pseudo-operator may occur that modifies a system property representing the effects of the event, such as the availability of some resource. This property could then be used as a precondition for other operators that may rely on that resource.

I believe an approach like this one is promising because it is simple, general, and easy to model. But this is just one possible way of modeling uncertainty, which I present in the interest of stimulating future research. Many other approaches are also possible and worthy of investigation.

A Case study interview protocol

This appendix reproduces the interview protocol used for the case study described in chapter 5, as approved by Carnegie Mellon University's institutional review board.

In a semistructured interview protocol, the interviewer (rather than rigidly adhering to a predefined schedule of questions) has the flexibility to explore participant responses by asking clarifying or follow-up questions on the spot. As a result, the exact set of questions cannot be known in advance. Rather, this protocol provides guidance on the overall form of the interview, the topics to be covered, and examples of key questions.

This protocol document is structured as a list of topics that we intend to cover in our interviews of software engineers at Costco. The questions listed under each topic are illustrative of the kinds of questions we plan to ask, but by no means exhaustive. In addition, the order of topics may be adjusted, or even entire topics excluded, depending on the individual being interviewed. (For example, a newly hired employee probably won't know much about the architecture evolution challenges that Costco has faced in the past, so questions on that topic would be skipped.) Obviously, some portions of the protocol are fixed: consent must always be obtained at the outset of the interview, and the "Conclusion" section will always be last.

A.1 Introductory consent script

My name is Jeffrey Barnes, and I am a PhD student working with Professor David Garlan at Carnegie Mellon University. We are conducting a research study to learn about the architectural evolution of software systems at Costco. As part of our information-gathering process, I would like to interview you to learn about Costco's software systems and software evolution practices. This interview will take approximately ____ minutes. Your participation in this research is completely voluntary, and you may withdraw from the study at any time.

With your permission, I will collect your name and contact information so that I can contact you later if I have follow-up questions. However, when we publish the results of this study, we will not use your name; the data that we obtain from this interview will be anonymized to protect your identity.

You should avoid using first names or specific names when providing information for this study.

Before we begin, I need to verify that you're eligible to participate in the study. Are you at least 18 years of age?

[If no, abort the interview.]

> Great. Do you have any questions about the study as I've described it?

[Answer any questions that the participant asks.]

> Do you consent to participate in this study?

[If no, abort the interview.]

> With your permission, I would like to take an audio recording of our conversation to ensure I capture it accurately. The audio recording will be kept private, so that only my research advisor and I have access to it. Is it OK if I record this interview?

[If no, conduct the interview without audio recording.]

A.2 Collection of personal identifiers

> I'd like to get your contact information so I can contact you later if I have any follow-up questions later or need to clarify something. Would you mind giving me your name, e-mail address, and phone number?

[Collect information from the participant. If the participant does not wish to provide complete contact information, but does wish to continue the interview, proceed without collecting information.]

A.3 The participant's role and background

Questions such as the following are necessary to understand the context for a participant's observations and descriptions.

- What's your job title?
- Can you explain what your role is within the organization?
- How long have you been working at Costco?
- What are the main software systems that you work on?
- What are the main kinds of tasks you're responsible for performing on a day-to-day basis?

A.4 Software architecture evolution at Costco

We will use questions such as the following to learn how software architecture evolution is planned and carried out at Costco today.

- As I mentioned earlier, my research is on the topic of software architecture evolution, so I’m interested in learning about how software engineers and architects at Costco plan and carry out major evolution of software systems. Can you tell me about any significant evolutions you’ve been involved with that would fit that description?
- Were the architectural changes necessary to achieve this evolution planned out in detail in advance, or were the changes instead made on an as-needed basis, without any overarching plan?
- What process was used to develop this evolution plan?
- Were any alternative plans considered?

A.5 Limitations of today’s approaches to software architecture evolution

With questions such as the following, we are seeking evidence supporting (or refuting) our hypothesis that today’s software architects are in need of better models and tools to help plan and carry out evolutions.

- What are some of the major challenges that you’ve faced in trying to plan or carry out the kinds of evolutions that we’ve been discussing?
- Do you feel that software architects could benefit from better tools for planning these kinds of evolutions, or do you think today’s approaches are generally adequate?
- Are there specific areas where you think better tool support would be especially helpful? Are there particular tasks involved in planning an architecture evolution that you think are especially suitable for automation?

A.6 Specifics about the evolution of particular software systems at Costco

In order to model Costco’s software systems accurately, we need to obtain specific information about the systems’ architectural structure and evolution. The following questions illustrate the kinds of questions we will ask, although the specific systems and components named in these sample questions are fictitious.

- Can you give me an overview of the architecture of the inventory management system?
- How does the ordering module interact with the billing subsystem?
- In which version of the system was the new database adapter you mentioned added?
- Were there any particular constraints you had to adhere to as you were restructuring this subsystem?

- What specific changes are planned for this system in the coming months?
- What are the main reasons for migrating this data store from MySQL to Amazon S3?

A.7 Conclusion

Those are all the questions I have for you today. Do you have any other questions for me about my research or this case study?

Here's my contact information in case you want to contact me regarding this research in the future. Thank you very much for your time.

B Content analysis coding guide

This appendix reproduces the coding frame used for the case study described in chapter 5. As explained in section 5.3.4, the content analysis in this case study was broken down into two separate content analyses, each with its own procedures and goals (and thus each with its own coding guide). The first content analysis addressed the "descriptive" questions of the case study, those that address how software architects plan and carry out evolution at Costco today. The second content analysis addressed the "evaluative" research question, which seeks to assess the suitability of our approach to architecture evolution by using the output of the content analysis for the construction of an evolution model.

B.1 Content analysis 1: Learning how architects do evolution

B.1.1 General principles

Each coding unit should be assigned exactly one category. No coding unit may be left uncategorized, and two categories may not be assigned to a single coding unit.

Top-level categories may not be assigned to coding units directly; instead, select the appropriate subcategory. That is, never assign the category "Approaches" to a coding unit. Instead, pick the appropriate subcategory, such as "Approaches: Phased development."

Each main category has a *residual* subcategory, for example "Approaches: Other." These residual categories should be used very sparingly if at all. Try to assign the best possible specific category to each coding unit, even if there is no perfect fit; only use the residual categories if there really is no suitable category.

In selecting a category for a given coding unit, consider the coding unit itself as well as its immediate context—up to a couple of paragraphs before and after the coding unit.

B.1.2 Categories

Evolution motives Abbreviation: mot	*Description:* Subcategories of this category should be applied to descriptions of impetuses that have motivated software evolution at Costco. These may be stated in terms of goals of the target system (e.g., an evolution occurs because the target system will have improved performance) or inadequacies of the initial system (e.g., an evolution occurs because the existing system has inadequate performance). This category may be applied regardless of whether the evolution described is one that has already happened, one that may yet occur, or one that was considered but not carried out.
Add features Abbreviation: mot-fea	*Description:* This category should be applied when an aim of evolution is to implement new functionality or new features. The line between a new feature and a quality improvement can sometimes be fuzzy. For example, an architect may describe a new feature that requires improvements to system interoperability to implement. The "Add features" subcategory should be used whenever a specific feature is being described; reserve the other subcategories, such as "Improve interoperability," for when the interviewee describes those goals as the primary reason for evolving. *Example:* "Currently we don't have any way to tell from looking at our pharmacy patient files, our optical patient files, our membership roll, our customers that we've reached out to try to sell memberships to, we don't have any way to say: Is the same person in there five times? There's no unique, one person we know, 'Hey, that's this guy here.' They need that."
Modernize technology Abbreviation: mot-mod	*Description:* This category should be applied when a stated aim of evolution is to adopt modern technology or abandon outdated technology. When outdated technology is discussed as a general challenge rather than as motivation for a specific evolution, use "Challenges: Dealing with legacy systems" instead. *Indicators:* The initial system is described as "dated," or irrelevant to "today's world," or the interviewee names a specific technology that an evolution aims to disuse. *Example:* "Eventually, I want this [mid-range system] stuff to all go away. I don't want to have to mess with [that] anymore"

Keep apace of business needs Abbreviation: mot-pac	*Description:* This category should be applied when an evolution is driven by the rapid pace of change within the industry or within the company. Note that the “Challenges” category has a similar subcategory: “Dealing with rapid change and anticipating the future.” When deciding between these two categories, consider whether the interviewee is describing a reason for a specific evolution or a general challenge that has arisen. *Indicators:* An evolution is described as being motivated by the “accelerating” pace of change, the “growing demand” of the business, or a need to “support the business curve.” *Example:* “The feeling was [. . .] that our current architecture and application systems would not scale to meet the growing demand of our growing business, because Costco’s an extreme-growth company.”
Improve flexibility Abbreviation: mot-flx	*Description:* This category should be applied when an aim of evolution is to make the system more flexible, or to ameliorate inflexibility in the current system. This category should be applied when flexibility in general is described as a goal—when an evolution is described as aiming to open up future possibilities generally. For the specific goal of increasing the system’s ability to interoperate or integrate with other systems, use the code “Evolution motives: Improve interoperability.” *Indicators:* An evolution is described as providing “flexibility,” making the system more “extensible,” or creating future “opportunities.” *Example:* “It was becoming quite brittle. And an opportunity had been presented to us to update to an object-oriented system that would provide some flexibility for anything that we might choose to do someday.”
Improve performance Abbreviation: mot-prf	*Description:* This category should be applied when an aim of evolution is to improve system performance. When performance is discussed as a general challenge rather than as motivation for a specific evolution, use “Challenges: Scalability, reliability, and performance” instead. *Example:* “We’re going to [a new point-of-sale package], which is basically the C++ version [. . .]. It’s supposed to be a little faster”

Improve reliability Abbreviation: mot-rel	*Description:* This category should be applied when an aim of evolution is to improve the availability, robustness, or resilience of the system. When reliability is discussed as a general challenge rather than as motivation for a specific evolution, use "Challenges: Scalability, reliability, and performance" instead. *Example:* "Now we're moving more into the highly available, somewhat more resilient message-based architecture, and I think that's going to be a good thing for us."
Improve interoperability Abbreviation: mot-xop	*Description:* This category should be applied when an aim of evolution is to make integration or interoperation among systems easier. When interoperability is discussed as a general challenge rather than as motivation for a specific evolution, use "Challenges: Communication, coordination, and integration challenges" instead. *Indicators:* The current system is described as being too "proprietary," or a stated goal is to make it easier to "hook to," "work with," "talk with," or "share information" with other systems. *Example:* "We're going to try and make those systems work with each other and talk with each other and share information and data back and forth."
Other Abbreviation: mot-?	*Description:* This category should be applied to evolution motives that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.
Causes of problems Abbreviation: cau	*Description:* Subcategories of this category should be applied to descriptions of circumstances that caused problems during the course of an evolution project. Architects often describe such circumstances in conjunction with their adverse effects, so "Causes of problems" codes and "Consequences" codes often occur in close proximity. Undesirable circumstances that have not caused any specifically articulated problems do not fall under this category but may fall under the "Challenges" category if they meet its criteria.

Lack of experience Abbreviation: cau-exp	*Description:* This category should be applied when a problem was caused by inexperience (or a failure to learn from experience), whether on the part of the architects planning an evolution or on the part of the engineers carrying it out. When inexperience is discussed as a challenge in general rather than as the cause of a specific adverse consequence, use "Challenges: Lack of expertise or experience" instead. *Example:* "The bulk of the people—especially, say, five years ago—that work in Costco IT—the bulk of them came up through our warehouses. So there's not a lot of ex-consultants running around here to have seen a million and one things. These are people who have seen one way of doing things. And then to say, you've got to do something new, and by the way you've never experienced that before, mistakes are going to get made."
Architects spread across the organization Abbreviation: cau-org	*Description:* This category should be applied when a problem was caused by architects being spread across many different groups within the organization, impeding communication among architects and complicating diagnosis of system failures. *Example:* "Our forms environment just couldn't work. Why? Because the architecture was spread across different groups"
Addressing too many problems up front Abbreviation: cau-prb	*Description:* This category should be applied when a problem was caused by attempting to solve all the problems of an evolution at its outset, rather than developing a basic plan that allows for adaptation to unforeseen circumstances. *Example:* "Let's just put in a foundational system that's open to the rest, rather than trying to solve all these questions at the time, which even now don't have answers for them. I think that was part of what happened in those multiple iterations as we tried to solve and could never get that compelling business reason down"
Cultural inertia Abbreviation: cau-cul	*Description:* This category should be applied when a problem was caused by cultural factors that inhibited change. *Example:* "When we started modernization, for whatever reason, we were talking more about COTS, but kind of in our DNA, we were still thinking *build*."
Forking an off-the-shelf package Abbreviation: cau-frk	*Description:* This category should be applied when a problem was caused by adopting and modifying an off-the-shelf software package, making it difficult to incorporate further updates from the vendor. *Example:* "We bought a package [. . .], and we did a very bad thing: we took the source code, we actually customized the source code for our need"

Ill-timed changes Abbreviation: cau-tim	*Description:* This category should be applied when a problem was caused by changing the system at an inopportune time. *Example:* "I don't know if you've ever heard of Finish Line; they're an athletic footwear dealer, online mainly. They piloted new e-comm and web software a week before Black Friday—the Friday after Thanksgiving. Not the best of times to pilot or make any changes to your website. It did not work too well."
Other Abbreviation: cau-?	*Description:* This category should be applied to causes that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.
Consequences Abbreviation: con	*Description:* Subcategories of this category should be applied to adverse consequences that arose due to missteps in an evolution project. Architects often describe problems in conjunction with their causes, so "Causes of problems" codes and "Consequences" codes often occur in close proximity.
Lost sales Abbreviation: con-sal	*Description:* This category should be applied when problems during an evolution resulted in financial loss. *Example:* "They figure it cost them about six million dollars in lost sales and things."
Wasted effort Abbreviation: con-eff	*Description:* This category should be applied when missteps during an evolution resulted in unnecessary effort or unnecessary complications, or when an evolution is rolled back and abandoned, or when the direction or basic approach of the evolution must be changed while it is being carried out due to unforeseen problems. *Example:* "We ran employees through literally tens of thousands of hours of Java training, but then they were not doing Java development."
Delays in evolving Abbreviation: con-dly	*Description:* This category should be applied when difficulties in planning an evolution delay the evolution's inception. *Example:* "I think that was part of what happened in those multiple iterations as we tried to solve and could never get that compelling business reason down, so it's taken so long for us to do this."
Limited upgradability Abbreviation: con-upg	*Description:* This category should be applied when missteps in an evolution result in a system that is difficult to upgrade further. *Example:* "[. . .] which rendered that package to be not upgradable for any future changes."

Other Abbreviation: con-?	*Description:* This category should be applied to consequences that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.
Challenges Abbreviation: cha	*Description:* Subcategories of this category should be applied to descriptions of challenges that Costco, or similar companies, or software organizations in general, face (or have faced in the past) when planning or carrying out evolution. This category should not be used for specific challenges that motivated evolution (these should instead be coded with subcategories of "Evolution motives"), nor for specific circumstances that caused problems in past efforts (these should be coded with subcategories of "Causes of problems").
Cultural challenges Abbreviation: cha-cul	*Description:* This category should be applied to descriptions of corporate-culture issues that present challenges, such as clashes between the corporate culture and the needs of the business, cultural resistance to needed innovations, or a need to contravene the organizational culture in order to accomplish goals. *Indicators:* The interviewee, in describing a challenge that Costco faces or has faced, uses words like "culture" or "cultural," or describes a tension between what the company says and what it does. *Example:* "Frankly that's been, I think, the biggest challenge of the modernization that we've undertaken, because [...] we have a culture of consensus here [...]. We don't do a lot of empowering some group to say how it's going to be, and then they just lay it out. That doesn't happen. Anything that we do has to be very collaborative if it's going to change the way people operate and whatnot. Frankly, we're still in the throes of getting that to work well."

Business justification Abbreviation: cha-bus	*Description:* This category should be applied to challenges in justifying architectural decisions (or the need for architecture, or the role of architects) in business terms (or to business executives or enterprise architects); challenges in managing the expectations of business executives; challenges caused by engineers having insufficient business knowledge to make good decisions; challenges in ensuring that products support business goals; and challenges in understanding business needs. *Indicators:* The interviewee discusses challenges relating to executive or business "buy-in" or "sponsorship," the difficulty of explaining architectural needs "to the business," business "concern" about architectural decision, or the attitudes of the "management," or the interviewee questions whether businesses are "ready" to do architecture, or the interviewee observes that engineers lack an "appreciation of the business." *Example:* "I think in terms of architecture, what I've seen the biggest challenges are executive buy-in on the architecture function in general, because when you throw architects in at the beginning (goodness, I hope it's at the beginning) of a project, you're adding time to it [...]. We've been fortunate that we've got CIO buy-in of what we're doing. But I think that's probably the largest obstacle, and when I meet with [architects] from other companies, they always say that that's the largest challenge they face."
Communication, coordination, and integration challenges Abbreviation: cha-com	*Description:* This category should be applied to challenges in integrating systems or facilitating communication among systems, to challenges of communication and coordination among people or teams, to challenges in managing multiple simultaneous initiatives, to challenges in appropriate use of documentation and standards, and to challenges of architecture conformance. The challenge of integrating with legacy systems specifically should instead be coded "Challenges: Dealing with legacy systems." When system integration is discussed as the impetus for a specific evolution rather than as a general challenge, use "Evolution motives: Improve interoperability." When organizational communication issues are discussed as the cause of a specific adverse consequence rather than as a general challenge, use "Causes of problems: Architects spread across the organization." *Example:* "One of the other challenges is just that we've grown now. We're in this building; the other half of IT is over on the corporate campus. You can't get all of IT together and have a presentation type of thing. [...] That's been a challenge. The communication is the biggest part."

Lack of expertise or experience Abbreviation: cha-exp	*Description:* This category should be applied to challenges arising from a lack of expertise or experience on the part of architects or engineers (or immaturity on the part of the organization as a whole), including general inexperience, a lack of knowledge about certain domains, or a lack of familiarity with particular tools. This category should also be applied to discussions of the difficulty of training people or the difficulty of learning new technologies. When inexperience directly causes a specifically articulated problem in the course of an evolution, use “Causes of problems: Lack of experience” instead. *Example:* “And the tools around that, really, we’re still very immature with.”
Dealing with rapid change and anticipating the future Abbreviation: cha-cha	*Description:* This category should be applied to challenges pertaining to planning ahead in the face of rapid change, seeing the long-term ramifications of decisions, dealing with extreme change within the company or throughout the industry, estimating future effort, adapting advances in the industry to Costco, or realizing a long-term vision. When rapid change is discussed as the impetus for a specific evolution rather than as a general challenge, use “Evolution motives: Keep apace of business needs” instead. *Indicators:* Instances of this category often refer to “extreme” or “massive” change; describe the company as moving at a very fast pace; or use metaphors such as shifting sand, a shifting landscape, or a narrow window of opportunity. *Example:* “The biggest challenge really, in my mind, is: How do you make sure that you are relevant? The wheel is moving, you know. Costco’s situation, I think, is one of the more extreme kind of conditions. We are trying to change so much in such a short time.”

Dealing with legacy systems Abbreviation: cha-leg	*Description:* This category should be applied to challenges that arise in dealing with legacy systems, including architectural problems with legacy systems, challenges in understanding legacy systems, challenges in bringing legacy vendor systems in-house, and difficulties in upgrading legacy systems (such as running a legacy system alongside a modernized system). When the difficulty of dealing with legacy systems is discussed as the impetus for a specific evolution rather than as a general challenge, use "Evolution motives: Modernize technology" instead. *Example:* "The applications that were built [...] here over time—over the twenty-five or thirty years or so that the midrange systems were here—really grew organically, and there wasn't comprehensive architectural thinking going on in the early days, to the point where you had a big ball of string. And now it's so brittle that [...] the simplest changes are taking months and months"
Scalability, reliability, and performance Abbreviation: cha-sca	*Description:* This category should be applied to challenges in managing large volumes of data; carrying out large implementations; meeting the needs of a company of Costco's size; ensuring transmission of data over the Internet; verifying the reliability of critical systems (e.g., through QA); and providing adequate system performance. When reliability or performance is discussed as the impetus for a specific evolution rather than as a general challenge, use "Evolution motives: Improve reliability" or "Evolution motives: Improve performance" instead. *Example:* "It's very limited to the options for a retailer of our size what we can use for that point-of-sale. Especially when the project started, and even somewhat now, Windows is always looked at as less than ideal as a stable platform for running point-of-sale, both from stability and uptime during the day, as well as the need for constant patches and those kind of things. So when you limit that away and look at other systems, other operating system types, to run it, there's even fewer that are out there."

Dealing with surprises Abbreviation: cha-sur	*Description:* This category should be applied to unexpected challenges that arise in the midst of an evolution, such as new requirements or unexpected infrastructure needs. If the unexpected challenge was a significant misstep that resulted in a specifically mentioned adverse consequence, use the appropriate subcategory of "Causes of problems" or "Consequences" instead. *Example:* "The reality is: things don't go as smooth as you'd like them to in the order you'd like them to. Sometimes development may be backed up, in which case you need to go renegotiate when does your project get started versus the projects that are in play. Capital expenditures: maybe we need to purchase extra infrastructure that we didn't realize needed to be purchased up front."
Managing scope and cost Abbreviation: cha-cst	*Description:* This category should be applied to challenges in scoping a project or keeping costs low. Challenges in reconciling large expenses with a cultural of frugality should instead be coded "Challenges: Cultural challenges." *Example:* "The biggest concern retailers have about [alternative payment methods] is the fees they have to pay as part of them, if you know what an interchange fee is. Basically there's a percentage you have to pay. [...] that's been a continuing challenge for the retailers."
Inadequate guidance and tool support Abbreviation: cha-gui	*Description:* This category should be applied when architects report that they do not have good resources to turn to for information, or that they have no good source of architectural guidance, or that they need better tools, or that available approaches are insufficient, or that they do not have enough guidance on projects from more senior people in the company. *Example:* "You can get shades-of-gray answers all over the place. Nobody's put a fine point on any aspect of it, so you can interpret what you will. I was telling somebody the other day, just go google *system context diagram*, and see how many flavors of that you get: like a thousand different models. [...] I think again, the challenge is there is no definitive resource."

Divergent understandings of architecture Abbreviation: cha-und	*Description:* This category should be applied to challenges that arise when there are people who have misunderstandings of architecture, disagreements about the role of architecture, or negative perceptions about architecture (or specific varieties of architecture). If the challenge as described is specific to dealing with business executives, use the category "Challenges: Business justification" instead. *Example:* "In general—not just Costco, but in general—there is a particular image that comes to a lot of people's mind when you say 'enterprise architecture.' Some people go into, 'Oh, man, that's a really needed thing that we've needed for a long time.' The other end of the spectrum is a bunch of ivory-tower impediments to progress."
Managing people Abbreviation: cha-ppl	*Description:* This category should be applied to challenges pertaining to the management of personnel, including dealing with employee uncertainty, retraining personnel with outdated expertise, motivating workers, and dealing with employee turnover. *Example:* "We're rooting out [our old] productivity suite, and we're going with [a different provider]. [...] Now, we had [administrators for the old system]. So what are you going to do? At first they were very resistant."
Managing the expectations and experience of users and stakeholders Abbreviation: cha-usr	*Description:* This category should be applied to challenges in managing users' and stakeholders' expectations about software systems or the software development process, as well as to challenges in ensuring that stakeholders and users of systems have a good experience. Included in this category are topics such as managing unrealistic expectations, understanding the political impacts of decisions, minimizing the impact of a system change on its users, and training users on a new system. *Example:* "The problem is in the user community, they feel like once they have your attention, they have to get everything they can out of you, otherwise they might not hear from you again for three years, and that's not going to help them. They come and they want the kitchen sink and everything, and it's very hard to tell them, we can't do that"
Other Abbreviation: cha-?	*Description:* This category should be applied to challenges that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.

Approaches Abbreviation: app	*Description:* Subcategories of this category should be applied to descriptions of approaches, methods, principles, structures, and tools that architects use (or have used, or have considered using) to help them plan and carry out evolutions. This includes everything from formal processes to very informal rules of thumb. It does not, however, include specific tactics or operations such as "introduce abstraction layer" or "wrap legacy component". Interviewees often spoke at considerable length and in significant detail about the approaches they use, to a much greater degree than is true for the other major categories in this coding frame. In addition, some of the subcategories of this category are rather broad topically. For example, "Challenges: Organizational strategies and structures" is a fairly broad topic, and interviewees often expound on the topic of organizational structure at considerable length. As a result, subcategories of this category will often apply to quite long segments of text, to a greater degree than is true of the other major categories, which usually apply to shorter passages, rarely longer than a few sentences.
Experience and intuition Abbreviation: app-int	*Description:* This category should be applied to descriptions of architects relying on their own experience, intuition, or judgment to make evolution decisions, or of architects learning from their experiences. *Example:* "I would say that my twenty-five years of experience in retail—I have a very unique perspective on what is realistic, what works, and how to stay away from the fringes and do what's right for the company."
Drawing from expert sources Abbreviation: app-exp	*Description:* This category should be applied to descriptions of drawing from expert sources to gain knowledge necessary to plan or carry out an evolution. Expert sources included experienced engineers within the company, outside consultants, product vendors, and books. *Example:* "Our current direction in Costco for these migrations between current state to these packaged solutions is leveraging a lot of external vendors [...] We are looking at them to provide. 'From [your] own best practices, how do you guys do that? We're not going to question you. You should tell us: How do you get this thing done, and how do you ensure there's knowledge transfer and support and all that stuff?'"

Industry practices Abbreviation: app-ind	*Description:* This category should be applied when Costco draws on the experiences and expertise of other companies in the industry. *Example:* "We've got a mobile app. We've got different types of mobile checkout people want to look at. How do we do that in the best way? So we're looking at what are the trends in the industry in mobile, both with our big competitors like Wal-Mart and Target, as well as [smaller chains]."
Phased development Abbreviation: app-pha	*Description:* This category should be used when the development or delivery of a project is broken into manageable phases or stages, or when an incremental or gradual transition strategy is used. *Example:* "I did a very iterative approach. I divided the project into three evolutions. Each evolution basically took on a chunk of our scope."
Tools Abbreviation: app-too	*Description:* This category should be applied to mentions of tools used in planning or carrying out evolution. (These need not be—and generally will not be—architecture evolution tools as such, but rather tools that architects and engineers use in the course of planning or carrying out evolution, including communication tools, code generation tools, reverse engineering tools, and process planning tools.) *Example:* "We got an analysis tool that we put onto our system and we had it do a drawing of all of the components and the relationships"
Formal approaches Abbreviation: app-frm	*Description:* This category should be applied to mentions of established processes, methods, and frameworks for software development, such as TOGAF and the Rational Unified Process. *Example:* "Rather than simply utilize EA group and the knowledge and experience we have [...], we're trying to use a couple additional tools, one of which is TOGAF. We're not strictly following that, but that's the basis of our architecture foundation. The core of that is the ADM process."
Prototypes, pilots, etc. Abbreviation: app-pil	*Description:* This category should be applied to descriptions of the use of prototypes, proofs of concept, pilots, reference applications, project templates, case studies, mock applications, and technical demonstrations for trying out or demonstrating innovations. *Example:* "For those architectural decisions, we develop guidance and references, and then further down we create a reference application that actually implements the decisions and the guidance that we are telling people to follow."

Training Abbreviation: app-trn	*Description:* This category should be applied to approaches for the training and mentoring of architects and engineers (including the retraining of engineers with unneeded skills). *Example:* "I went through the ISA training and got certified on that"
Business architecture Abbreviation: app-bus	*Description:* This category should be applied to descriptions of the use of business architecture practices, or the consideration of business concerns in planning architectural operations more generally, or thinking in terms of business capabilities. *Example:* "Instead of all these very virtual or kind of theoretical categories to rank a service, we just look at business capability. Services are supposed to be aligned with the business when we build a service, so why don't we just go straight to the source? So what we ended up doing is looking at business capability from the business architecture team."
Communication and coordination practices Abbreviation: app-com	*Description:* This category should be applied to descriptions of approaches for facilitating communication and coordination within the organization, including coordinating efforts, allocating tasks, providing guidance, documenting decisions, communicating with stakeholders, and incorporating feedback. When the discussion centers on the challenges of communication and coordination rather than approaches for communication and coordination, use the category "Challenges: Communication, coordination, and integration challenges" instead. *Example:* "It's really more about communication than architecting systems. That collaboration aspect, I think, is absolutely paramount to the success of an architect. You have to talk to lots of people all the time. The drawings don't have to be that precise, as long as they communicate the information."
Considering alternatives Abbreviation: app-alt	*Description:* This category should be applied when architects consider alternative plans or backup plans for evolving a system. *Example:* "We have to always, I think, have plan A and plan B. Going into a lot of the work that we're doing now, I say, well, I would love to have this SOA service environment, but understanding that that's still very much in its infancy, here I have plan B, which is let's just continue on with what we know using legacy systems and legacy technologies."

Anticipating the future Abbreviation: app-fut	*Description:* This category should be applied to descriptions of anticipating future developments as an approach for better planning evolution. When anticipating the future is cited as a challenge rather than as a strategy, use the category "Challenge: Dealing with rapid change and anticipating the future" instead. *Example:* "Besides point-of-sale, we're modernizing our loyalty system, our CRM system. We're modernizing all those parts and pieces. As we modernize those, then we look to that future where they all have an interaction with each other."
Rules of thumb and informal strategies Abbreviation: app-rot	*Description:* This category should be applied whenever an interviewee articulates a general rule of thumb or informal strategy for planning or carrying out evolution, such as "lean toward simplicity," "prefer open standards," or "identify areas of uncertainty." *Example:* "Consistency is very important in my book. I'm less about which standard's a better standard than the other—rather that these have to be enterprise standards—everybody should do it the same way."
Organizational strategies and structures Abbreviation: app-org	*Description:* This category should be applied to descriptions of the organizational structures and strategies that Costco uses in architecting systems. This includes discussions of architects' and engineers' roles and the structure, function, and formation of groups and teams. *Example:* "On that EA side, as we talked about, there's the domain architects, which span each pillar: information, integration, business, mobility, security, ... Large number of people over on that side. We're supposed to be focused, as SA, on the implementation or the project level, and they're supposed to be focused on more of the strategy and the domain expertise for us to go to for implementation to make sure we're using best practices in the security domain, for example, or infrastructure domain, or some domain that an individual architect may not be versed in"
Process Abbreviation: app-pro	*Description:* This category should be applied to descriptions of software process at Costco, including descriptions of stages of software development and discussions of project life cycle. *Example:* "There's this SDM (solution delivery methodology) out there, which is basically the end-to-end life cycle of all the people involved in a project. Start-up is the first phase; that's when you engage the PM and do as we talked about, the pre–requirements gathering. Then it goes to solution outline, which finalizes in the EARB. From there you go to macro design. Then you go to micro design, implementation, deployment, and close-down."

Other Abbreviation: app-?	*Description:* This category should be applied to approaches that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.

B.2 Content analysis 2: Modeling a real-world evolution

B.2.1 General principles

Each coding unit should be assigned exactly one subcategory of *each* applicable main category. (Note that this differs from the procedure in content analysis 1, in which each coding unit should be assigned only one category *total*.) The "Classification" main category is applicable to all coding units. Thus, all coding units should be assigned exactly one subcategory of the "Classification" main category. The other main categories are applicable to coding units that describe architectural elements. Thus, coding units that have been classified as components, connectors, ports or roles, systems, or groupings should be assigned exactly one subcategory of "Presence in initial architecture" and one subcategory of "Presence in target architecture." Those that have been classified as containment relations, evolution operations, evolution constraints, dimensions of concerns, or "Other" should not be coded with respect to the "Presence in initial architecture" and "Presence in target architecture" categories.

Because the coding units in this content analysis are not contiguous—each coding unit consists of a set of isolated fragments of content spread throughout the data—substantial surrounding context may be necessary to accurately categorize a coding unit. This surrounding context certainly extends to each page on which the coding unit occurs in the architectural documentation, as well as a couple of paragraphs before and after each occurrence of the coding unit in the interview transcripts. In a few cases where categorization is particularly difficult, it may be appropriate to go beyond the research data and consider external sources of information. Specifically, the following sources may be considered if necessary to accurately judge the proper categorization of a coding unit:

- The socket server diagram that I received in connection with the interviews
- The cash recyler integration architecture document that I received in connection with the interviews
- Publicly available documentation pertaining to specific commercially available software products, obtained from vendor websites (in the case of coding units corresponding to off-the-shelf products)

However, the architectural documentation and interview transcripts that form the research data should be the principal consideration. The external documents listed above should be used only as necessary to settle close calls in cases of ambiguity.

B.2.2 Categories

Classification Abbreviation: c	*Description:* Subcategories of this category are used to identify what type of entity (software element, constraint, or dimension of concern) a coding unit refers to. Every coding unit should fall into one of these subcategories; therefore, each coding unit should be assigned exactly one subcategory of this category. If a coding unit fails to correspond to any of the subcategories, categorize it as "Classification: Other." If there appear to be multiple subcategories that could apply to a coding unit, read the descriptions of the subcategories below carefully for discussion of the distinctions among them, so that you can choose the single most appropriate classification.
Component Abbreviation: c-cmp	*Description:* This category should be applied to software elements that are best characterized as *components*. A component represents a computational element within a software system. Examples of kinds of components include software packages, services, data stores, and data processing elements. A component has a well-defined identity, a well-defined boundary, and a well-defined interface by which it interoperates with other software elements. The "Component" category may be confused with several other categories, notably "Connector," "System," and "Grouping." See the definitions of these other categories below for discussion of the distinctions. *Indicators:* A software element that is described as an "application," "controller," "file," "handler," "log," "product," or "server" is usually a component. However, many components are not described in such terms, and a few elements that *are* described in these terms are better classified as systems or connectors. Thus, components must ultimately be identified based on their general characteristics, as described above. An off-the-shelf package should always be classified as a component (unless it exists chiefly to facilitate communication, in which case it is a connector). In graphical diagrams, components are often depicted as boxes. However, boxes are used for a great many other purpose, including representation of connectors, systems, and groupings, so this is by no means a reliable indicator of a component.

Connector Abbreviation: c-cnn	*Description:* This category should be applied to software elements that are best characterized as *connectors.* A connector represents a pathway of interaction between two or more components. Examples of kinds of connectors include pipes, streams, buses, message queues, and other kinds of communication channels. Although connectors are often thought of as simple, some connectors are quite complex, and connectors can even have an internal substructure, just as components can. Thus, do not categorize an element as a component simply because it seems complex and bulky. Instead, the determination of whether an element is a component or a connector should be guided by its function. If it is principally a computational or data-processing element, it should be categorized as a component. If its principal role is to facilitate communication between components, it should be categorized as a connector. *Indicators:* A software element that is described as a "bridge," "broker," "bus," "queue," or "transfer" is almost certainly a connector. However, connectors are not always described in such terms and often must be identified based on their general characteristics, as described above. In prose (spoken or written), connectors are often not discussed in explicit terms as first-class entities, but instead appear implicitly as relationships between components. For example, if a component is described as "talking to" another component, the phrase "talking to" is evidence of a connector between the two components. A coding unit that captures a relation between components (or systems or groupings) should always be classified as either a connector or a containment relation. In graphical diagrams, components are often depicted as lines (or arrows). In fact, a line between two components in a diagram almost always represents a connector. However, the converse is not true. Connectors can be diagrammatically represented in many other ways besides as lines—including as boxes—so the fact that an element appears as something other than a line in a diagram is not evidence that it is not a connector.

Port or role Abbreviation: c-att	*Description:* This category should be applied to coding units that express an attachment relation between a component (or system or grouping) and a connector. (This could be characterized architecturally as a port, a role, or a combination of a port attached to a role.) Application of this category is relatively straightforward. When a coding unit expresses a relationship between a component (or system or grouping) and connector, that unit should be either coded as "Port or role" (if the relationship is one of component-connector attachment) or as "Containment relation" (if the relationship is one of containment).

System Abbreviation: c-sys	*Description:* This category should be applied when a coding unit refers to a complete software system. The distinction between a system and a component can be hazy, since components may themselves be fairly large and complex and may contain other software elements. However, a software system is understood to operate as a complete and relatively independent whole (although it may have dependencies on other systems with which it interoperates), while a component is intended to operate as one piece of a larger system. In some cases, though, the distinction between a system and a component is merely a matter of perspective. For example, an off-the-shelf point-of-sale package could certainly be considered a system from the perspective of its developers, the package vendor, but from Costco's perspective, it is just one component of Costco's point-of-sale system. Thus, this application should be coded as a component rather than a system, since we are taking the perspective of Costco. Topologically, systems are equivalent to components. Thus, connectors, which normally connect components to components, may also connect systems to systems, or components to systems. Indeed, a system may be regarded as a type of component. The distinction is made in this coding frame simply because it facilitates understanding of system boundaries and areas of responsibility. Also murky is the distinction between a system and a grouping. In general, a system is a concrete package of software elements that have been deliberately composed together into a unified whole. The *grouping* category below is for more nebulous collections of elements. Such groupings may be used to associate elements logically, to express physical boundaries, or to demark software layers or tiers. *Indicators:* Systems are often easy to recognize simply because they are called "systems"—for example, the point-of-sale *system*, or the membership *system*. However, this indicator is not foolproof; the term *system* sometimes appears in descriptions of things that might be better categorized as components. Notably, off-the-shelf packages should always be classified as components (or connectors), even though they might be regarded as systems under other circumstances.

Grouping Abbreviation: c-grp	*Description:* This category should be applied to coding units that express logical groupings of software elements, such as software layers or tiers. It should also be applied to coding units that express the physical boundaries within which software elements may be contained. For example, architects frequently distinguish between software elements that are contained within individual Costco warehouses, and those that are part of the central corporate infrastructure. In this case, *warehouse* defines a physical boundary and should be considered a grouping. Sometimes it may be unclear whether to categorize a unit as a grouping or a component. Components, after all, can (like groupings) contain other software elements. However, components exist as specifically identifiable elements within the software; they are discrete elements with a well-defined interface, intended to interoperate with other software elements. Groupings are more nebulous, and are often purely logical; in other words, it may be useful to use them to group related software elements together for purposes of discussion and analysis, but they may not have any real presence in the software. It may also be possible to confuse groupings with systems. See the definition of the "System" category above for a discussion of the distinction. Groupings, like systems, are topologically equivalent to components. See the description of the "System" category for an explanation of this point.
Containment relation Abbreviation: c-cnt	*Description:* This category should be applied to coding units that express a containment relation between two software elements—that is, when a coding unit expressing a relationship between two software elements implies that one of them is contained within the other. *Indicators:* In diagrams, containment is easy to recognize because it is almost always represented by physical containment; for example, one box (representing a system) contains another, smaller box (representing a component). Containment relations are not so clearly manifest in prose. However, a containment relation is often described in terms of the contained element being "in" the containing element, or the containing element "having" the contained element.

Evolution operation Abbreviation: c-eop	*Description:* This category should be applied to coding units that describe an operation that may be carried out during the evolution. Evolution operations can be simple—such as adding, removing, or modifying individual software elements—or moderately complex, such as interposing a bridging element between two components to facilitate communication, or replacing a network of point-to-point connectors with a bus.
Evolution constraint Abbreviation: c-cns	*Description:* This category should be applied to coding units that express a constraint on the evolution of the system. Examples of kinds of evolution constraints including ordering constraints on evolution operations, architectural constraints characterizing the structure that the system must have at specific points within the evolution, timing constraints expressing when operations must be carried out, and organizational constraints expressing which organizational elements carry out which tasks. Constraints should be easy to distinguish from all of the other categories here except perhaps “Dimension of concern.” After all, constraints and dimensions of concern both represent qualities that are desired of an evolution. The difference is that constraints express compulsory requirements that an evolution must have in order to be valid. Dimensions of concern represent qualities that lie along a range or spectrum, with one end of the range being preferable to the other, and points in the middle of the range being intermediate between them. Thus, “availability” generally is a dimension of concern, while “The system must maintain 99% availability at all times” is a constraint.
Dimension of concern Abbreviation: c-cnc	*Description:* This category should be applied to coding units that capture a dimension of concern relevant to the evolution of the system—something that could, at least in principle, be quantified and used as an optimization criterion in planning the evolution. Examples of possible dimensions of concern include cost, effort, evolution duration, and architectural qualities such as performance or reliability. For the difference between constraints and dimensions of concern, see the definition of the “Evolution constraint” category above.
Other Abbreviation: c-?	*Description:* This category should be applied to coding units that do not fall into any of the above categories. Categorize coding unit in the most appropriate category above if at all possible; use this residual category only if none of the above categories are at all applicable.

Presence in initial architecture Abbreviation: init	*Description:* Subcategories of this category are used to indicate whether a software element is present in the initial architecture of the system. Note that only coding units corresponding to software elements should be assigned a subcategory of this category. Coding units not corresponding to software elements (i.e., coding units tagged “c-cnt,” “c-eop,” “c-cns,” “c-cnc,” or “c-?”) should not be assigned a subcategory of this category, since these concepts (constraints, operations, etc.) are not local to individual evolution states. However, any coding unit that has been identified as a software element (i.e., a coding unit classified “c-cmp,” “c-cnn,” “c-att,” “c-sys,” or “c-grp”) should be assigned exactly one subcategory of this category. The “initial architecture” in this case is the structure of the system at the outset of the point-of-sale evolution, before any of the modernizations described in the material had been effected.
Present Abbreviation: init-p	*Description:* This category should be applied when the software element is present in the initial architecture, at the outset of the evolution.
Absent Abbreviation: init-a	*Description:* This category should be applied when the software element is not yet present in the initial architecture, at the outset of the evolution.
Presence in target architecture Abbreviation: targ	*Description:* Subcategories of this category are used to indicate whether a software element is present in the initial architecture of the system. The same rules apply here as to the “Presence in initial architecture category”: software elements should be assigned exactly one subcategory, and other coding units should not be assigned any. The “target architecture” is the structure of the system at the conclusion of the point-of-sale evolution, once all currently planned changes have been made. (However, speculative changes that are described as possibly occurring in the distant future, but for which no concrete plans currently exist, should not be considered as part of this evolution.)
Present Abbreviation: targ-p	*Description:* This category should be applied when the software element is present in the target architecture, at the conclusion of the evolution.
Absent Abbreviation: targ-a	*Description:* This category should be applied when the software element is no longer present in the target architecture, at the conclusion of the evolution.

C PDDL specification

This appendix reproduces the PDDL specification used in the automation work described in section 6.3. As explained in section 6.3.1, a PDDL specification comprises two parts: a domain description and a problem description.

C.1 Domain description

```
(define (domain data-center-migration)
(:requirements :typing :durative-actions :action-costs :duration-inequalities)

(:types
    DataCenter Host Service Day - object
    UnixHost WindowsHost - Host)

(:constants
    DC1 DC2 - DataCenter
    Monday Tuesday Wednesday Thursday Friday Saturday Sunday - Day)

(:predicates
    (is-in ?h - Host ?dc - DataCenter)
    (is-on ?s - Service ?h - Host)
    (has-firewall ?dc - DataCenter)

    (network-switch-installed)

    (was-removed-from ?h - Host ?dc - DataCenter)
    (was-migrated ?s - Service)
    (not-yet-migrated ?s - Service)
    (can-be-migrated-individually ?s - Service)
    (is-unused ?h - Host)

    (ok-to-move-on ?s - Service ?d - Day)

    (no-work-in-progress)

    (next ?d1 ?d2 - Day)
    (weekend ?d - Day)
    (today ?d - Day))

(:functions
    (total-cost)
    (current-hour)
    (next-day)
    (time-since-last-day)
    (cost-multiplier ?d - Day)

    (uid ?s - Service)
    (allowed-downtime ?s - Service)
    (service-count ?h - Host))

(:durative-action waitTillNextDay
    :parameters (?oldDay ?newDay - Day)
    :duration (= ?duration (- next-day current-hour))
    :condition (and
        (at start (no-work-in-progress))
        (at start (today ?oldDay))
        (over all (next ?oldDay ?newDay))
        (at start (<= current-hour next-day)))
    :effect (and
        (at start (not (no-work-in-progress)))
        (at end (no-work-in-progress))
        (at end (not (today ?oldDay)))
        (at end (today ?newDay))
        (at end (increase (current-hour) ?duration))
        (at end (increase (next-day) 8))
        (at end (assign (time-since-last-day) 0))))

(:durative-action installSwitch
    :parameters (?d - Day)
    :duration (= ?duration 1.9)
    :condition (and
        (at start (no-work-in-progress))
        (over all (today ?d))
        (at start (<= time-since-last-day 6.1)))
    :effect (and
        (at start (not (no-work-in-progress)))
        (at end (no-work-in-progress))
        (at end (network-switch-installed))
        (at start (increase (total-cost) (* 1 (cost-multiplier ?d))))
        (at end (increase current-hour 1.9))
        (at end (increase time-since-last-day 1.9))))

(:durative-action installFirewall
    :parameters (?dc - DataCenter ?d - Day)
    :duration (= ?duration 0.9)
    :condition (and
        (at start (no-work-in-progress))
        (over all (today ?d))
        (at start (<= time-since-last-day 7.1)))
    :effect (and
        (at start (not (no-work-in-progress)))
        (at end (no-work-in-progress))
        (at end (has-firewall ?dc))
        (at start (increase (total-cost) (* 5 (cost-multiplier ?d))))
        (at end (increase current-hour 0.9))
        (at end (increase time-since-last-day 0.9))))

(:durative-action cloneHost1
    :parameters (?h1 ?h2 - UnixHost ?s - Service ?d - Day)
    :duration (= ?duration 1.9)
    :condition (and
        (at start (is-on ?s ?h1))
        (over all (is-in ?h1 DC1))
        (over all (is-in ?h2 DC2))
        (over all (has-firewall DC2))
        (over all (network-switch-installed))
        (at start (not-yet-migrated ?s))
        (at start (is-unused ?h2))
        (at start (no-work-in-progress))
        (over all (today ?d))
        (at start (<= time-since-last-day 6.1))
        (over all (= (service-count ?h1) 1)))
    :effect (and
        (at end (is-on ?s ?h2))
        (at end (was-migrated ?s))
        (at end (not (not-yet-migrated ?s)))
        (at end (not (is-unused ?h2)))
        (at start (not (no-work-in-progress)))
        (at end (no-work-in-progress))
        (at start (increase (total-cost) (* 5 (cost-multiplier ?d))))
        (at end (increase current-hour 1.9))
        (at end (increase time-since-last-day 1.9))))

(:durative-action cloneHost2
    :parameters (?h1 ?h2 - UnixHost ?s1 ?s2 - Service ?d - Day)
    :duration (= ?duration 1.9)
    :condition (and
        (at start (is-on ?s1 ?h1))
        (at start (is-on ?s2 ?h1))
        (over all (is-in ?h1 DC1))
        (over all (is-in ?h2 DC2))
        (over all (has-firewall DC2))
        (over all (network-switch-installed))
```

```
            (at start (not-yet-migrated ?s1))
            (at start (not-yet-migrated ?s2))
            (at start (is-unused ?h2))
            (at start (no-work-in-progress))
            (over all (today ?d))
            (at start (<= time-since-last-day 6.1))
            (over all (< (uid ?s1) (uid ?s2))))
        :effect (and
            (at end (is-on ?s1 ?h2))
            (at end (is-on ?s2 ?h2))
            (at end (was-migrated ?s1))
            (at end (was-migrated ?s2))
            (at end (not (not-yet-migrated ?s1)))
            (at end (not (not-yet-migrated ?s2)))
            (at end (not (is-unused ?h2)))
            (at start (not (no-work-in-progress)))
            (at end (no-work-in-progress))
            (at start (increase (total-cost) (* 5 (cost-multiplier ?d))))
            (at end (increase current-hour 1.9))
            (at end (increase time-since-last-day 1.9))))

    (:durative-action manuallyMigrateService
        :parameters (?s - Service ?h1 ?h2 - Host ?d - Day)
        :duration (= ?duration 3.9)
        :condition (and
            (at start (is-on ?s ?h1))
            (over all (is-in ?h1 DC1))
            (over all (is-in ?h2 DC2))
            (over all (has-firewall DC2))
            (over all (network-switch-installed))
            (at start (not-yet-migrated ?s))
            (over all (can-be-migrated-individually ?s))
            (over all (ok-to-move-on ?s ?d))
            (at start (no-work-in-progress))
            (over all (today ?d))
            (over all (>= (allowed-downtime ?s) 3.9))
            (at start (<= time-since-last-day 4.1)))
        :effect (and
            (at end (is-on ?s ?h2))
            (at end (was-migrated ?s))
            (at end (not (not-yet-migrated ?s)))
            (at end (not (is-unused ?h2)))
            (at start (not (no-work-in-progress)))
            (at end (no-work-in-progress))
            (at start (increase (total-cost) (* 20 (cost-multiplier ?d))))
        (at end (increase current-hour 3.9))
        (at end (increase time-since-last-day 3.9))))

(:durative-action physicallyMoveHost1
    :parameters (?h - Host ?s - Service ?d - Day)
    :duration (= ?duration 5.9)
    :condition (and
        (at start (is-in ?h DC1))
        (over all (is-on ?s ?h))
        (over all (has-firewall DC2))
        (over all (network-switch-installed))
        (at start (not-yet-migrated ?s))
        (over all (ok-to-move-on ?s ?d))
        (at start (no-work-in-progress))
        (over all (today ?d))
        (over all (= (service-count ?h) 1))
        (over all (>= (allowed-downtime ?s) 5.9))
        (at start (<= time-since-last-day 2.1)))
    :effect (and
        (at end (is-in ?h DC2))
        (at end (not (is-in ?h DC1)))
        (at end (was-removed-from ?h DC1))
        (at end (was-migrated ?s))
        (at end (not (not-yet-migrated ?s)))
        (at start (not (no-work-in-progress)))
        (at end (no-work-in-progress))
        (at start (increase (total-cost) (* 50 (cost-multiplier ?d))))
        (at end (increase current-hour 5.9))
        (at end (increase time-since-last-day 5.9))))

(:durative-action physicallyMoveHost2
    :parameters (?h - Host ?s1 ?s2 - Service ?d - Day)
    :duration (= ?duration 5.9)
    :condition (and
        (at start (is-in ?h DC1))
        (over all (is-on ?s1 ?h))
        (over all (is-on ?s2 ?h))
        (over all (has-firewall DC2))
        (over all (network-switch-installed))
        (at start (not-yet-migrated ?s1))
        (at start (not-yet-migrated ?s2))
        (over all (ok-to-move-on ?s1 ?d))
        (over all (ok-to-move-on ?s2 ?d))
        (at start (no-work-in-progress))
        (over all (today ?d))
        (at start (<= time-since-last-day 2.1))
```

```
            (over all (>= (allowed-downtime ?s1) 5.9))
            (over all (>= (allowed-downtime ?s2) 5.9))
            (over all (< (uid ?s1) (uid ?s2))))
        :effect (and
            (at end (is-in ?h DC2))
            (at end (not (is-in ?h DC1)))
            (at end (was-removed-from ?h DC1))
            (at end (was-migrated ?s1))
            (at end (was-migrated ?s2))
            (at end (not (not-yet-migrated ?s1)))
            (at end (not (not-yet-migrated ?s2)))
            (at start (not (no-work-in-progress)))
            (at end (no-work-in-progress))
            (at start (increase (total-cost) (* 50 (cost-multiplier ?d))))
            (at end (increase current-hour 5.9))
            (at end (increase time-since-last-day 5.9))))

    (:durative-action decommissionHost
        :parameters (?h - Host ?dc - DataCenter ?d - Day)
        :duration (= ?duration 3.9)
        :condition (and
            (over all (network-switch-installed))
            (at start (no-work-in-progress))
            (over all (today ?d))
            (at start (<= time-since-last-day 4.1)))
        :effect (and
            (at end (not (is-in ?h ?dc)))
            (at end (was-removed-from ?h ?dc))
            (at start (not (no-work-in-progress)))
            (at end (no-work-in-progress))
            (at start (increase (total-cost) (* 10 (cost-multiplier ?d))))
            (at end (increase current-hour 3.9))
            (at end (increase time-since-last-day 3.9))))

    )
```

C.2 Problem description

```
    (define (problem data-center-migration)
    (:domain data-center-migration)

    (:objects
        ClientWebsiteHost1 ClientWebsiteHost2 SafetyDbHost
            UnusedUnixHost1 UnusedUnixHost2 UnusedUnixHost3 - UnixHost
        FinanceHost AnalyticsHost UnusedWindowsHost1 UnusedWindowsHost2
            - WindowsHost
    ClientWebsiteService1 ClientWebsiteService2 ClientWebsiteService3
        ClientWebsiteService4 SafetyDbService PayrollService
        AccountingService AnalyticsService - Service)

(:init
    (is-in ClientWebsiteHost1 DC1)
    (is-in ClientWebsiteHost2 DC1)
    (is-in SafetyDbHost DC1)
    (is-in FinanceHost DC1)
    (is-in AnalyticsHost DC1)
    (is-in UnusedUnixHost1 DC2)
    (is-in UnusedUnixHost2 DC2)
    (is-in UnusedUnixHost3 DC2)
    (is-in UnusedWindowsHost1 DC2)
    (is-in UnusedWindowsHost2 DC2)

    (is-on ClientWebsiteService1 ClientWebsiteHost1)
    (is-on ClientWebsiteService2 ClientWebsiteHost1)
    (is-on ClientWebsiteService3 ClientWebsiteHost2)
    (is-on ClientWebsiteService4 ClientWebsiteHost2)
    (is-on SafetyDbService SafetyDbHost)
    (is-on PayrollService FinanceHost)
    (is-on AccountingService FinanceHost)
    (is-on AnalyticsService AnalyticsHost)

    (has-firewall DC1)

    (not-yet-migrated ClientWebsiteService1)
    (not-yet-migrated ClientWebsiteService2)
    (not-yet-migrated ClientWebsiteService3)
    (not-yet-migrated ClientWebsiteService4)
    (not-yet-migrated SafetyDbService)
    (not-yet-migrated PayrollService)
    (not-yet-migrated AccountingService)
    (not-yet-migrated AnalyticsService)

    (can-be-migrated-individually ClientWebsiteService1)
    (can-be-migrated-individually ClientWebsiteService2)
    (can-be-migrated-individually ClientWebsiteService3)
    (can-be-migrated-individually ClientWebsiteService4)
    (can-be-migrated-individually SafetyDbService)
    (can-be-migrated-individually PayrollService)
    (can-be-migrated-individually AccountingService)

    (is-unused UnusedUnixHost1)
    (is-unused UnusedUnixHost2)
    (is-unused UnusedUnixHost3)
```

```
(ok-to-move-on ClientWebsiteService1 Monday)
(ok-to-move-on ClientWebsiteService1 Tuesday)
(ok-to-move-on ClientWebsiteService1 Wednesday)
(ok-to-move-on ClientWebsiteService1 Thursday)
(ok-to-move-on ClientWebsiteService1 Friday)
(ok-to-move-on ClientWebsiteService1 Saturday)
(ok-to-move-on ClientWebsiteService1 Sunday)
(ok-to-move-on ClientWebsiteService2 Monday)
(ok-to-move-on ClientWebsiteService2 Tuesday)
(ok-to-move-on ClientWebsiteService2 Wednesday)
(ok-to-move-on ClientWebsiteService2 Thursday)
(ok-to-move-on ClientWebsiteService2 Friday)
(ok-to-move-on ClientWebsiteService2 Saturday)
(ok-to-move-on ClientWebsiteService2 Sunday)
(ok-to-move-on ClientWebsiteService3 Monday)
(ok-to-move-on ClientWebsiteService3 Tuesday)
(ok-to-move-on ClientWebsiteService3 Wednesday)
(ok-to-move-on ClientWebsiteService3 Thursday)
(ok-to-move-on ClientWebsiteService3 Friday)
(ok-to-move-on ClientWebsiteService3 Saturday)
(ok-to-move-on ClientWebsiteService3 Sunday)
(ok-to-move-on ClientWebsiteService4 Monday)
(ok-to-move-on ClientWebsiteService4 Tuesday)
(ok-to-move-on ClientWebsiteService4 Wednesday)
(ok-to-move-on ClientWebsiteService4 Thursday)
(ok-to-move-on ClientWebsiteService4 Friday)
(ok-to-move-on ClientWebsiteService4 Saturday)
(ok-to-move-on ClientWebsiteService4 Sunday)
(ok-to-move-on SafetyDbService Monday)
(ok-to-move-on SafetyDbService Tuesday)
(ok-to-move-on SafetyDbService Wednesday)
(ok-to-move-on SafetyDbService Thursday)
(ok-to-move-on SafetyDbService Friday)
(ok-to-move-on SafetyDbService Saturday)
(ok-to-move-on SafetyDbService Sunday)
(ok-to-move-on PayrollService Tuesday)
(ok-to-move-on PayrollService Wednesday)
(ok-to-move-on PayrollService Thursday)
(ok-to-move-on PayrollService Friday)
(ok-to-move-on PayrollService Saturday)
(ok-to-move-on PayrollService Sunday)
(ok-to-move-on AccountingService Saturday)
(ok-to-move-on AccountingService Sunday)
(ok-to-move-on AnalyticsService Monday)
```

```
(ok-to-move-on AnalyticsService Tuesday)
(ok-to-move-on AnalyticsService Wednesday)
(ok-to-move-on AnalyticsService Thursday)
(ok-to-move-on AnalyticsService Friday)
(ok-to-move-on AnalyticsService Saturday)
(ok-to-move-on AnalyticsService Sunday)

(no-work-in-progress)

(next Monday Tuesday)
(next Tuesday Wednesday)
(next Wednesday Thursday)
(next Thursday Friday)
(next Friday Saturday)
(next Saturday Sunday)
(next Sunday Monday)
(weekend Saturday)
(weekend Sunday)
(today Monday)

(= (total-cost) 0.0)
(= (current-hour) 0.0)
(= (next-day) 8)
(= (time-since-last-day) 0)

(= (cost-multiplier Monday) 1)
(= (cost-multiplier Tuesday) 1)
(= (cost-multiplier Wednesday) 1)
(= (cost-multiplier Thursday) 1)
(= (cost-multiplier Friday) 1)
(= (cost-multiplier Saturday) 3)
(= (cost-multiplier Sunday) 3)

(= (uid ClientWebsiteService1) 0)
(= (uid ClientWebsiteService2) 1)
(= (uid ClientWebsiteService3) 2)
(= (uid ClientWebsiteService4) 3)
(= (uid SafetyDbService) 4)
(= (uid PayrollService) 5)
(= (uid AccountingService) 6)
(= (uid AnalyticsService) 7)

(= (allowed-downtime ClientWebsiteService1) 4.0)
(= (allowed-downtime ClientWebsiteService2) 4.0)
(= (allowed-downtime ClientWebsiteService3) 4.0)
(= (allowed-downtime ClientWebsiteService4) 4.0)
(= (allowed-downtime SafetyDbService) 0.0)
```

```
        (= (allowed-downtime PayrollService) 160.0)
        (= (allowed-downtime AccountingService) 16.0)
        (= (allowed-downtime AnalyticsService) 168.0)

        (= (service-count ClientWebsiteHost1) 2)
        (= (service-count ClientWebsiteHost2) 2)
        (= (service-count SafetyDbHost) 1)
        (= (service-count FinanceHost) 2)
        (= (service-count AnalyticsHost) 1))

    (:goal
        (and

            ; All services have been migrated.
            (forall (?s - Service) (was-migrated ?s))

            ; No hosts remain in DC1.
            (was-removed-from ClientWebsiteHost1 DC1)
            (was-removed-from ClientWebsiteHost2 DC1)
            (was-removed-from SafetyDbHost DC1)
            (was-removed-from FinanceHost DC1)
            (was-removed-from AnalyticsHost DC1)))

    (:metric minimize (total-cost)))
```

Bibliography

I have employed a couple of conventions to make this bibliography easier to navigate. First, in the margin alongside each entry, I have listed the page numbers on which citations of the given work appear. This makes it easy to find where in this dissertation a particular work is cited.

Second, I have typeset my own name in bold wherever it appears within this bibliography. This makes it easy to find previous publications that I have authored or coauthored on the topic of software architecture evolution.

[1] T. K. Abdel-Hamid (1989). "The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach." *IEEE Transactions on Software Engineering* 15: 109–119. doi:10.1109/32.21738 p. 146

[2] M. Abi-Antoun, J. Aldrich (2009). "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure Using Annotations." Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Languages and Applications (OOPSLA), pp. 321–340. ACM. ISBN 978-1-60558-734-9. doi:10.1145/1640089.1640113 p. 159

[3] M. Ali Babar, J. M. Verner, P. T. Nguyen (2007). "Establishing and Maintaining Trust in Software Outsourcing Relationships: An Empirical Investigation." *Journal of Systems and Software* 80(9): 1438–1449. doi:10.1016/j.jss.2006.10.038 p. 67

[4] R. Alur, T. A. Henzinger (1989). "A Really Temporal Logic." Proceedings of the Symposium on Foundations of Computer Science (FOCS), pp. 164–169. IEEE. ISBN 0-8186-1982-1. doi:10.1109/SFCS.1989.63473 pp. 29, 32

[5] P. America et al. (2011). "Architecting for Improved Evolvability." In *Views on Evolvability of Embedded Systems* (eds. P. van de Laar, T. Punter), chap. 2, pp. 21–36. Springer. ISBN 978-90-481-9848-1. doi:10.1007/978-90-481-9849-8_2 p. 147

[6] A. I. Antón, C. Potts (2003). "Functional Paleontology: The Evolution of User-Visible System Services." *IEEE Transactions on Software Engineering* 29(2): 151–166. doi:10.1109/TSE.2003.1178053 p. 151

[7] S. Anwar, M. Ramzan, A. Rauf, A. A. Shahid (2010). "Software Maintenance Prediction Using Weighted Scenarios: An Architecture Perspective." Proceedings of the International Conference on Information Science & Applications (ICISA). IEEE. ISBN 978-1-4244-5943-8. doi:10.1109/ICISA.2010.5480420 p. 148

[8] F. Bachmann, L. Bass, M. Klein (2003). "Deriving Architectural Tactics: A Step Toward Methodical Architectural Design." Tech. Rep. CMU/SEI-2003-TR-004, Software Engineering Institute, Pittsburgh. http://www.sei.cmu.edu/reports/03tr004.pdf p. 149

[9] N. Baddoo, T. Hall (2002). "Motivators of Software Process Improvement: An Analysis of Practitioners' Views." *Journal of Systems and Software* 62(2): 85–96. doi:10.1016/S0164-1212(01)00125-X p. 67

[10] R. Bahsoon, W. Emmerich (2003). "ArchOptions: A Real Options-Based Model for Predicting the Stability of Software Architectures." Proceedings of the International Workshop on Economic-Driven Software Engineering Research (EDSER). http://www.soberit.hut.fi/edser-5/Papers/E03_BahsoonEmmerich.pdf p. 148

[11] C. Y. Baldwin, K. B. Clark (1999). *Design Rules*, vol. 1. MIT. ISBN 0-262-02466-7. p. 145

[12] M. S. Ball, G. W. H. Smith (1992). *Analyzing Visual Data*. Sage. ISBN 0-8039-3434-3. p. 77

[13] O. Barais, A. F. Le Meur, L. Duchien, J. Lawall (2008). "Software Architecture Evolution." In *Software Evolution* (eds. T. Mens, S. Demeyer), pp. 233–262. Springer. ISBN 978-3-540-76439-7. doi:10.1007/978-3-540-76440-3_10 pp. 148, 149

[14] O. Barais et al. (2004). "TranSAT: A Framework for the Specification of Software Architecture Evolution." Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT), pp. 31–38. ISBN 84-688-6782-9. pp. 148, 149

[15] **J. M. Barnes** (2011). "NASA's Advanced Multimission Operations System: A Case Study in Formalizing Software Architecture Evolution." Cleared Document 11-3608, Jet Propulsion Laboratory, Pasadena, CA. http://hdl.handle.net/2014/42232 p. 41

[16] **J. M. Barnes** (2012). "NASA's Advanced Multimission Operations System: A Case Study in Software Architecture Evolution." Proceedings of the International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA), pp. 3–12. ACM. ISBN 978-1-4503-1346-9. doi:10.1145/2304696.2304700 pp. 41, 115

[17] **J. M. Barnes** (2013). "Case Study Report: Architecture Evolution at Costco." Tech. Rep. CMU-ISR-13-116, Carnegie Mellon University, Pittsburgh. http://reports-archive.adm.cs.cmu.edu/anon/isr2013/CMU-ISR-13-116.pdf ch. 5 passim

[18] **J. M. Barnes**, D. Garlan (2013). "Challenges in Developing a Software Architecture Evolution Tool as a Plug-In." Proceedings of the Workshop on Developing Tools as Plug-Ins (TOPI), pp. 13–18. IEEE. ISBN 978-1-4673-6288-7. doi:10.1109/TOPI.2013.6597188 pp. 119, 143

[19] **J. M. Barnes**, D. Garlan, B. Schmerl (in press). "Evolution Styles: Foundations and Models for Software Architecture Evolution." *Software and Systems Modeling*. doi:10.1007/s10270-012-0301-9 pp. 11, 13, 15, 25, 81, 165

[20] **J. M. Barnes**, A. Pandey, D. Garlan (2013). "Automated Planning for Software Architecture Evolution." Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). To appear. pp. 125, 143

[21] L. Bass, P. Clements, R. Kazman (2003). *Software Architecture in Practice*. Addison–Wesley, 2nd edn. ISBN 0-321-15495-9. p. 2

[22] A. Bauer, M. Leucker, C. Schallhart (2011). "Runtime Verification for LTL and TLTL." *ACM Transactions on Software Engineering and Methodology* 20(4): article 14. doi:10.1145/2000799.2000800 p. 32

[23] P. Bell (2001). "Content Analysis of Visual Images." In *Handbook of Visual Analysis* (eds. T. van Leeuwen, C. Jewitt), pp. 10–34. Sage. ISBN 0-7619-6476-2. p. 77

[24] P. Bengtsson, J. Bosch (1999). "Architecture Level Prediction of Software Maintenance." Proceedings of the European Conference on Software Maintenance & Reengineering (CSMR), pp. 139–147. IEEE. ISBN 0-7695-0090-0. doi:10.1109/CSMR.1999.756691 p. 148

[25] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet (2004). "Architecture-Level Modifiability Analysis (ALMA)." *Journal of Systems & Software* 69: 129–147. doi:10.1016/S0164-1212(03)00080-3 pp. 89, 148

[26] J. Benton, A. Coles, A. Coles (2012). "Temporal Planning with Preferences and Time-Dependent Continuous Costs." Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 2–10. AAAI. http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699 p. 139

[27] B. Berelson (1952). *Content Analysis in Communication Research*. Free Press. pp. 67, 68

[28] W. Bischofberger, J. Kühl, S. Löffler (2004). "Sotograph—A Pragmatic Approach to Source Code Architecture Conformance Checking." Proceedings of the European Workshop on Software Architecture (EWSA), LNCS, vol. 3047, pp. 1–9. Springer. ISBN 978-3-540-24769-2. doi:10.1007/978-3-540-24769-2_1 p. 159

[29] P. Blackburn (2000). "Internalizing Labelled Deduction." *Journal of Logic and Computation* 10(1): 137–168. doi:10.1093/logcom/10.1.137 p. 30

[30] P. Blackburn, M. Tzakova (1999). "Hybrid Languges and Temporal Logic." *Logic Journal of the IGPL* 7(1): 27–54. doi:10.1093/jigpal/7.1.27 p. 30

[31] B. Boehm (1981). *Software Engineering Economics*. Prentice Hall. ISBN 0-13-822122-7. p. 146

[32] B. Boehm (1996). "Anchoring the Software Process." *IEEE Software* 13(4): 73–82. doi:10.1109/52.526834 p. 146

[33] B. Boehm, C. Abts, S. Chulani (2000). "Software Development Cost Estimation Approaches – A Survey." *Annals of Software Engineering* 10: 177–205. doi:10.1023/A:1018991717352 p. 146

[34] M. Boyle, S. Shannon (2011). "Wal-Mart Brings Back 8,500 Products in Bid to End U.S. Slump." *Bloomberg.com*. http://www.bloomberg.com/news/2011-04-11/wal-mart-brings-back-8-500-products-in-bid-to-end-u-s-slump.html p. 62

[35] L. Bratthall (2001). *Empirical Studies of the Impact of Architectural Understanding on Software Evolution*. Ph.D. thesis, University of Oslo. pp. 152, 153

[36] R. Brcina, M. Riebisch (2008). "Architecting for Evolvability by Means of Traceability and Features." Proceedings of the International ERCIM Workshop on Software Evolution and Evolvability (Evol), pp. 72–81. IEEE. ISBN 978-1-4244-2776-5. doi:10.1109/ASEW.2008.4686323 p. 148

[37] H. P. Breivold, I. Crnkovic, R. Land, M. Larsson (2008). "Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study." Proceedings of the International Conference on Software Engineering Advances (ICSEA), pp. 205–213. IEEE. ISBN 978-0-7695-3372-8. doi:10.1109/ICSEA.2008.16 p. 148

[38] H. P. Breivold, I. Crnkovic, M. Larsson (2012). "A Systematic Review of Software Architecture Evolution Research." *Information and Software Technology* 54(1): 16–40. doi:10.1016/j.infsof.2011.06.002 p. 145

[39] N. Brown, R. L. Nord, I. Ozkaya, M. Pais (2011). "Analysis and Management of Architectural Dependencies in Iterative Release Planning." Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 103–112. IEEE. ISBN 978-0-7695-4351-2. doi:10.1109/WICSA.2011.22 pp. 21, 150, 153

[40] M. Bucholtz (2000). "The Politics of Transcription." *Journal of Pragmatics* 32(10): 1439–1465. doi:10.1016/S0378-2166(99)00094-6 p. 66

[41] Y. Cai, K. J. Sullivan (2005). "Simon: Modeling and Analysis of Design Space Structures." Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 329–332. ACM. ISBN 1-58113-993-4. doi:10.1145/1101908.1101962 p. 148

[42] Y. Cai, K. J. Sullivan (2006). "Modularity and Analysis of Logical Design Models." Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 91–102. IEEE. ISBN 0-7695-2579-2. doi:10.1109/ASE.2006.53 p. 148

[43] E. G. Carmines, R. A. Zeller (1979). *Reliability and Validity Assessment.* No. 07-017 in Quantitative Applications in the Social Sciences, Sage. ISBN 0-8039-1371-0. p. 110

[44] C. Cassell, G. Symon (1994). "Qualitative Research in Work Contexts." In *Qualitative Methods in Organizational Research: A Practical Guide* (eds. C. Cassell, G. Symon), pp. 1–13. Sage. ISBN 0-8039-8769-2. p. 68

[45] A. Cervantes (2009). "Exploring the Use of a Test Automation Framework." Proceedings of the IEEE Aerospace Conference. IEEE. ISBN 978-1-4244-2622-5. doi:10.1109/AERO.2009.4839695 p. 44

[46] S. Chaki, N. Sharygina, N. Sinha (2004). "Verification of Evolving Software." Proceedings of the Workshop on Specification and Verification of Component Based Systems (SAVCBS), pp. 55–61. Iowa State University. http://www.eecs.ucf.edu/~leavens/SAVCBS/2004/savcbs04.pdf p. 145

[47] Y. Chan, N. Ivanov, O. Mueller (2013). *Oracle to DB2 Conversion Guide: Compatibility Made Easy.* Redbooks, IBM, 3rd edn. ISBN 0-7384-3875-8. p. 152

[48] N. Chapin et al. (2001). "Types of Software Evolution and Software Maintenance." *Journal of Software Maintenance and Evolution: Research and Practice* 13(1): 3–30. doi:10.1002/smr.220 p. 91

[49] S.-W. Cheng, D. Garlan (2012). "Stitch: A Language for Architecture-Based Self-Adaptation." *Journal of Systems and Software* 85(12): 2860–2875. doi:10.1016/j.jss.2012.02.060 p. 169

[50] S.-W. Cheng et al. (2002). "Using Architectural Style as a Basis for System Self-Repair." Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 45–59. Kluwer. ISBN 1-4020-7176-0. p. 149

[51] L. Chung, K. Cooper, A. Yi (2003). "Developing Adaptable Software Architectures Using Design Patterns: An NFR Approach." *Computer Standards & Interfaces* 25: 253–260. doi:10.1016/S0920-5489(02)00096-X p. 147

[52] E. M. Clarke, E. A. Emerson, A. P. Sistla (1986). "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications." *ACM Transactions on Programming Languages and Systems* 8(2): 244–263. doi:10.1145/5397.5399 p. 32

[53] P. Clements et al. (2011). *Documenting Software Architectures: Views and Beyond.* Addison–Wesley, 2nd edn. ISBN 0-321-55268-7. pp. 12, 13, 109, 165

[54] J. Cohen (1960). "A Coefficient of Agreement for Nominal Scales." *Educational and Psychological Measurement* 20(1): 37–46. doi:10.1177/001316446002000104 p. 102

[55] S. A. Conger (1994). *The New Software Engineering.* Wadsworth. ISBN 0-534-17143-5. p. 146

[56] J.-R. C. Cook (2010). "Engineers Diagnosing Voyager 2 Data System." Press Release 2010-151, Jet Propulsion Laboratory, Pasadena, CA. http://www.jpl.nasa.gov/news/news.cfm?release=2010-151 p. 41

[57] G. A. Cornstock, E. A. Rubinstein, eds. (1972). *Television and Social Behavior: Reports and Papers*, vol. I. National Institute of Mental Health. http://www.eric.ed.gov/ERICWebPortal/detail?accno=ED059623 p. 67

[58] Costco (2012). "Form 10-K for the Fiscal Year Ended September 2, 2012." SEC Accession No. 0001193125-12-428890. http://edgar.secdatabase.com/1732/119312512428890/filing-main.htm p. 62

[59] Costco (2013). "Form 10-Q for the Quarterly Period Ended May 12, 2013." SEC Accession No. 0001445305-13-001463. http://edgar.secdatabase.com/2430/144530513001463/filing-main.htm pp. 61, 62

[60] G. Cowan, M. O'Brien (1990). "Gender and Survival vs. Death in Slasher Films: A Content Analysis." *Sex Roles* 23(3/4): 187–196. doi:10.1007/BF00289865 p. 77

[61] S. Cresswell, A. Coddington (2004). "Compilation of LTL Goal Formulas into PDDL." Proceedings of the European Conference on Artificial Intelligence (ECAI), FAIA, vol. 110, pp. 985–986. IOS. ISBN 978-1-58603-452-8. http://frontiersinai.com/ecai/ecai2004/ecai04/p0983.html p. 131

[62] D. Crockford (2006). "The application/json Media Type for JavaScript Object Notation (JSON)." RFC 4627, IETF. http://www.ietf.org/rfc/rfc4627 p. 16

[63] C. E. Cuesta, E. Navarro, D. E. Perry, C. Roda (2013). "Evolution Styles: Using Architectural Knowledge as an Evolution Driver." *Journal of Software: Evolution and Process* 25(9): 957–980. doi:10.1002/smr.1575 p. 150

[64] W. Cunningham (1992). "The WyCash Portfolio Management System." Addendum to the Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 29–30. ACM. ISBN 0-89791-610-7. doi:10.1145/157709.157715 p. 170

[65] W. A. Cushing (2012). *When is Temporal Planning* Really *Temporal?* Ph.D. thesis, Arizona State University. http://sagarmatha.eas.asu.edu/cushing-dissertation.pdf p. 141

[66] K. Czarnecki, S. Helsen (2006). "Feature-Based Survey of Model Transformation Approaches." *IBM Systems Journal* 45(3): 621–645. doi:10.1147/sj.453.0621 p. 169

[67] N. Dehghani (2007). "Mission Data Processing and Control Subsystem (MPCS)." Presented at the European Ground System Architecture Workshop (ESAW). http://www.egos.esa.int/export/egos-web/others/Events/Workshop/ESAW-workshop-2007/day2-Session-4-0900-1040/04_Dehghani.pdf pp. 44, 50

[68] N. Dehghani, Q. Sun, M. Demore (2009). "NASA's 2011 Mars Science Laboratory (MSL) & Supporting Ground Data System Architecture." Presented at the European Ground System Architecture Workshop (ESAW). http://www.egos.esa.int/export/egos-web/others/Events/Workshop/ESAW-workshop-2009/Day1-Session-1-0920-1055/S01_03_Dehghani.pdf pp. 44, 50

[69] N. Dehghani, M. Tankenson (2006). "A Multi-mission Event-Driven Component-Based System for Support of Flight Software Development, ATLO, and Operations First Used by the Mars Science Laboratory (MSL) Project." Cleared Document 06-0909, Jet Propulsion Laboratory, Pasadena, CA. Presented at the AIAA International Conference on Space Operations (SpaceOps 2006). http://hdl.handle.net/2014/39852 pp. 43, 44, 50

[70] C. Del Rosso, A. Maccari (2007). "Assessing the Architectonics of Large, Software-Intensive Systems Using a Knowledge-Based Approach." Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), paper 2. IEEE. ISBN 0-7695-2744-2. doi:10.1109/WICSA.2007.17 pp. 89, 148

[71] S. Demri, R. Lazić, D. Nowak (2007). "On the Freeze Quantifier in Constraint LTL: Decidability and Complexity." *Information and Computation* 205(1): 2–24. doi:10.1016/j.ic.2006.08.003 pp. 29, 30

[72] S. Demri, R. Lazić, A. Sangnier (2010). "Model Checking Memoryful Linear-Time Logics over One-Counter Automata." *Theoretical Computer Science* 411(22–24): 2298–2316. doi:10.1016/j.tcs.2010.02.021 p. 37

[73] N. K. Denzin, Y. S. Lincoln (2011). "Introduction: The Discipline and Practice of Qualitative Research." In *The Sage Handbook of Qualitative Research* (eds. N. K. Denzin, Y. S. Lincoln), pp. 1–19. Sage, 4th edn. ISBN 978-1-4129-7417-2. p. 68

[74] H. Dixon et al. (2008). "Portrayal of Tanning, Clothing Fashion and Shade Use in Australian Women's Magazines, 1987–2005." *Health Education Research* 23(5): 791–802. doi:10.1093/her/cym057 p. 77

[75] S. Ducasse, D. Pollet (2009). "Software Architecture Reconstruction: A Process-Oriented Taxonomy." *IEEE Transactions on Software Engineering* 35: 573–591. doi:10.1109/TSE.2009.19 p. 2

[76] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian (2008). "Selecting Empirical Methods for Software Engineering Research." In *Guide to Advanced Empirical Software Engineering* (eds. F. Shull, J. Singer, D. I. K. Sjøberg), pp. 285–311. Springer. ISBN 978-1-84800-043-8. doi:10.1007/978-1-84800-044-5_11 p. 7

[77] Ecma International (2011). *Standard ECMA-262: ECMAScript Language Specification*, 5.1 edn. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf p. 21

[78] S. Edelkamp, J. Hoffmann (2004). "PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition." Tech. Rep. 195, Department of Computer Science, University of Freiburg. http://tr.informatik.uni-freiburg.de/2004/Report195/ p. 127

[79] A. Engel, T. R. Browning (2008). "Designing Systems for Adaptability by Means of Architecture Options." *Systems Engineering* 11: 125–146. doi:10.1002/sys.20090 p. 148

[80] M. Erder, P. Pureur (2006). "Transitional Architectures for Enterprise Evolution." *IT Pro* 8(3): 10–17. doi:10.1109/MITP.2006.77 p. 151

[81] H. Fahmy, R. C. Holt (2000). "Using Graph Rewriting to Specify Software Architectural Transformations." Proceedings of the IEEE International Conference on Automated Software Engineering (ASE), pp. 187–196. IEEE. ISBN 0-7695-0710-7. doi:10.1109/ASE.2000.873663 p. 149

[82] L. Fischman, K. McRitchie, D. D. Galorath (2005). "Inside SEER-SEM." *CrossTalk: The Journal of Defense Software Engineering* 18(4): 26–28. http://www.crosstalkonline.org/storage/issue-archives/2005/200504/200504-Fischman.pdf p. 146

[83] M. Fitting (2002). "Modal Logics between Propositional and First-Order." *Journal of Logic and Computation* 12(6): 1017–1026. doi:10.1093/logcom/12.6.1017 p. 29

[84] J. L. Fleiss, B. Levin, M. C. Paik (2003). *Statistical Methods for Rates and Proportions.* Wiley, 3rd edn. ISBN 0-471-52629-0. pp. 102, 104

[85] R. Flesch (1948). "A New Readability Yardstick." *Journal of Applied Psychology*, 32(3). doi:10.1037/h0057532 p. 67

[86] M. Fox, D. Long (2002). "PDDL+: Modelling Continuous Time-Dependent Effects." Proceedings of the International NASA Workshop on Planning and Scheduling for Space. p. 141

[87] M. Fox, D. Long (2003). "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains." *Journal of Artificial Intelligence Research* 20: 61–124. doi:10.1613/jair.1129 p. 127

[88] M. Franceschet, M. de Rijke (2006). "Model Checking Hybrid Logics (with an Application to Semistructured Data)." *Journal of Applied Logic* 4(3): 279–304. doi:10.1016/j.jal.2005.06.010 p. 34

[89] J. Frank, K. Golden, A. Jonsson (2003). "The Loyal Opposition Comments on Plan Domain Description Languages." Proceedings of the ICAPS Workshop on PDDL. http://users.cecs.anu.edu.au/~thiebaux/workshops/ICAPS03/proceedings/frank.pdf p. 131

[90] D. G. Freelon (2010). "ReCal: Intercoder Reliability Calculation as a Web Service." *International Journal of Internet Science* 5(1): 20–33. http://www.ijis.net/ijis5_1/ijis5_1_freelon_pre.html p. 103

[91] E. Fricke, A. P. Schulz (2005). "Design for Changeability (DfC): Principles to Enable Changes in Systems Throughout Their Entire Lifecycle." *Systems Engineering* 8: 342–359. doi:10.1002/sys.20039 p. 147

[92] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley. ISBN 0-201-63361-2. p. 145

[93] M. R. Garey, D. S. Johnson (1979). *Computers and Intractability*. Freeman. ISBN 0-7167-1044-7. p. 37

[94] D. Garlan, **J. M. Barnes**, B. Schmerl, O. Celiku (2009). "Evolution Styles: Foundations and Tool Support for Software Architecture Evolution." Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA), pp. 131–140. IEEE. ISBN 978-1-4244-4984-2. doi:10.1109/WICSA.2009.5290799 p. 11

[95] D. Garlan, R. Monroe, D. Wile (1997). "Acme: An Architecture Description Interchange Language." Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), pp. 169–183. ACM. doi:10.1145/1925805.1925814 pp. 12, 78

[96] D. Garlan, B. Schmerl (2009). "Ævol: A Tool for Defining and Planning Architecture Evolution." Proceedings of the International Conference on Software Engineering (ICSE), pp. 591–594. IEEE. ISBN 978-1-4244-3452-7. doi:10.1109/ICSE.2009.5070563 pp. 118, 142, 143

[97] A. Gerevini, D. Long (2006). "Preferences and Soft Constraints in PDDL3." Proceedings of the ICAPS Workshop on Planning with Preferences and Soft Constraints, pp. 46–53. http://strathprints.strath.ac.uk/3149/ p. 127

[98] A. Gerevini, A. Saetti, I. Serina (2006). "An Approach to Temporal Planning and Scheduling in Domains with Predictable Exogenous Events." *Journal of Artificial Intelligence Research* 25: 187–231. doi:10.1613/jair.1742 p. 139

[99] M. Ghallab, D. Nau, P. Traverso (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann. ISBN 1-55860-856-7. p. 126

[100] C. Ghezzi, M. Jazayeri, D. Mandrioli (1991). *Fundamentals of Software Engineering*. Prentice Hall. ISBN 0-13-820432-2. p. 145

[101] S. S. Gokhale (2007). "Architecture-Based Software Reliability Analysis: Overview and Limitations." *IEEE Transactions on Dependable and Secure Computing* 4(1): 32–40. doi:10.1109/TDSC.2007.4 p. 86

[102] V. Goranko (1994). "Temporal Logic with Reference Pointers." Proceedings of the International Conference on Temporal Logic (ICTL), LNCS, vol. 827, pp. 133–148. Springer. ISBN 3-540-58241-X. doi:10.1007/BFb0013985 pp. 29, 31

[103] W. B. Green (1995). "Multimission Ground Data System Support of NASA's Planetary Program." *Acta Astronautica* 37: 407–415. doi:10.1016/0094-5765(95)00067-A p. 42

[104] S. Gregor (2006). "The Nature of Theory in Information Systems." *MIS Quarterly* 30(3): 611–642. p. 7

[105] N. Groeben, R. Rustemeyer (1994). "On the Integration of Quantitative and Qualitative Methodological Paradigms (Based on the Example of Content Analysis)." In *Trends and Perspectives in Empirical Social Research* (eds. I. Borg, P. P. Mohler), pp. 308–326. Walter de Gruyter. ISBN 3-11-014312-7. p. 71

[106] L. Grunske (2005). "Formalizing Architectural Refactorings as Graph Transformation Systems." Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing & International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN), pp. 324–329. IEEE. ISBN 0-7695-2294-7. doi:10.1109/SNPD-SAWN.2005.37 pp. 148, 149

[107] K. L. Gwet (2012). *Handbook of Inter-Rater Reliability*. Advanced Analytics, 3rd edn. ISBN 978-0-9708062-7-7. p. 104

[108] T. A. Henzinger (1990). "Half-Order Modal Logic: How to Prove Real-Time Properties." Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), pp. 281–296. ACM. ISBN 0-89791-404-X. doi:10.1145/93385.93429 p. 26

[109] ISO (2006). *International Standard ISO/IEC 14764: Software Engineering—Software Life Cycle Processes—Maintenance*. 2nd edn. ISBN 0-7381-4961-6. p. 90

[110] ISO (2012). *International Standard ISO/IEC 19506: Information Technology—Object Management Group Architecture-Driven Modernization (ADM)—Knowledge Discovery Meta-Model (KDM)*. p. 151

[111] A. Jaffe (2012). "Transcription in Practice: Nonstandard Orthography." In *Orthography as Social Action: Scripts, Spelling, Identity and Power* (eds. A. Jaffe, J. Androutsopoulos, M. Sebba, S. Johnson), pp. 203–224. De Gruyter. ISBN 978-1-61451-136-6. p. 66

[112] P. Jamshidi, M. Ghafari, A. Ahmad, C. Pahl (2013). "A Framework for Classifying and Comparing Architecture-Centric Software Evolution Research." Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 305–314. IEEE. ISBN 978-0-7695-4948-4. doi:10.1109/CSMR.2013.39 pp. 91, 145

[113] Jet Propulsion Laboratory (2010). "Voyager: Spacecraft Lifetime." http://voyager.jpl.nasa.gov/spacecraft/spacecraftlife.html p. 42

[114] Jet Propulsion Laboratory (2011). "AMMOS." http://ammos.jpl.nasa.gov/ p. 42

[115] Jet Propulsion Laboratory (2013). "Missions." http://www.jpl.nasa.gov/missions/ p. 42

[116] S. Kang, D. Garlan (in press). "Architecture-Based Planning of Software Evolution." *International Journal of Software Engineering and Knowledge Engineering*. p. 150

[117] J. Kirk, M. L. Miller (1986). *Reliability and Validity in Qualitative Research*. Sage. ISBN 0-8039-2560-3. p. 110

[118] H. Koziolek (2011). "Sustainability Evaluation of Software Architectures: A Systematic Review." Proceedings of the Joint ACM SIGSOFT Conference on the Quality of Software Architectures & ACM SIGSOFT Symposium on Architecting Critical Systems (QoSA+ISARCS), pp. 3–12. ACM. ISBN 978-1-4503-0724-6. doi:10.1145/2000259.2000263 p. 147

[119] S. Kracauer (1952). "The Challenge of Qualitative Content Analysis." *Public Opinion Quarterly* 16(4): 631–642. doi:10.1086/266427 p. 68

[120] K. Krippendorff (1970). "Bivariate Agreement Coefficients for Reliability Data." *Sociological Methodology* 2: 139–150. p. 102

[121] K. Krippendorff (2004). "Reliability in Content Analysis: Some Common Misconceptions and Recommendations." *Human Communication Research* 30(3): 411–433. doi:10.1111/j.1468-2958.2004.tb00738.x p. 103

[122] K. Krippendorff (2013). *Content Analysis: An Introduction to Its Methodology*. Sage, 3rd edn. ISBN 1-4129-8315-0. ch. 5 passim

[123] P. Kruchten, R. L. Nord, I. Ozkaya, D. Falessi (2013). "Technical Debt: Towards a Crisper Definition." *ACM SIGSOFT Software Engineering Notes* 38(5): 51–54. doi:10.1145/2507288.2507326 p. 170

[124] S. C. Kurtik, J. B. Berner, M. Levesque (2000). "Calling Home in 2003: JPL Roadmap to Standardized TT&C Customer Support." Cleared Document 00-0675, Jet Propulsion Laboratory, Pasadena, CA. Presented at the International Space Ops 2000 Symposium. http://hdl.handle.net/2014/14262 p. 43

[125] S. Kvale, S. Brinkmann (2009). *InterViews: Learning the Craft of Qualitative Research Interviewing*. Sage, 2nd edn. ISBN 978-0-7619-2542-2. pp. 65, 111

[126] L. Lamport (1994). "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems* 16(3): 872–923. doi:10.1145/177492.177726 p. 29

[127] N. Lassing, D. Rijsenbrij, H. van Vliet (1999). "Towards a Broader View on Software Architecture Analysis of Flexibility." Proceedings of the Asia Pacific Software Engineering Conference (APSEC), pp. 238–245. IEEE. ISBN 0-7695-0509-0. doi:10.1109/APSEC.1999.809608 p. 89

[128] E. Laursen (2011). "High-End Trading Strategists See Cost Savings in Cloud Computing." *Institutional Investor*. http://www.institutionalinvestor.com/Popups/PrintArticle.aspx?ArticleID=2750046 p. 4

[129] O. Le Goaer, P. Ebraert (2007). "Evolution Styles: Change Patterns for Software Evolution." Proceedings of the International ERCIM Symposium on Software Evolution (Evol), pp. 252–261. ftp://ftp.umh.ac.be/pub/ftp_infofs/2007/ERCIM-Evol2007.pdf p. 149

[130] O. Le Goaer, M.-C. Oussalah, D. Tamzalit, A.-D. Seriai (2008). "Evolution Shelf: Exploiting Evolution Styles within Software Architectures." Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 387–392. Knowledge Systems Institute. ISBN 1-891706-22-5. http://green.cs.ksi.edu/conference/Proceedings/seke/SEKE2008_Proceedings.pdf p. 149

[131] O. Le Goaer, D. Tamzalit, M. Oussalah (2010). "Evolution Styles to Capitalize Evolution p. 149

Expertise within Software Architectures.” Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 159–164. Knowledge Systems Institute. ISBN 1-891706-26-8. http://green.cs.ksi.edu/conference/Proceedings/seke/SEKE2010_Proceedings.pdf

[132] O. Le Goaer, D. Tamzalit, M. Oussalah, A.-D. Seriai (2008). “Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures.” Proceedings of the International Computer Software & Applications Conference (COMPSAC), pp. 311–318. IEEE. ISBN 978-0-7695-3262-2. doi:10.1109/COMPSAC.2008.104 pp. 149, 153

[133] O. Le Goaer, D. Tamzalit, M. C. Oussalah, A.-D. Seriai (2008). “Evolution Styles to the Rescue of Architectural Evolution Knowledge.” Proceedings of the International Workshop on Sharing and Reusing Architectural Knowledge (SHARK), pp. 31–36. ACM. ISBN 978-1-60558-038-8. doi:10.1145/1370062.1370071 p. 149

[134] M. D. LeCompte, L. P. Goetz (1982). “Problems of Reliability and Validity in Ethnographic Research.” *Review of Educational Research* 52(1): 31–60. doi:10.3102/00346543052001031 p. 111

[135] M. M. Lehman (1980). “On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle.” *Journal of Systems and Software* 1: 213–221. doi:10.1016/0164-1212(79)90022-0 p. 145

[136] M. M. Lehman (1980). “Programs, Life Cycles, and Laws of Software Evolution.” *Proceedings of the IEEE* 68(9): 1060–1076. doi:10.1109/PROC.1980.11805 p. 145

[137] M. M. Lehman (1996). “Laws of Software Evolution Revisited.” Proceedings of the European Workshop on Software Process Technology (EWSPT), LNCS, vol. 1149, pp. 108–124. Springer. ISBN 3-540-61771-X. doi:10.1007/BFb0017737 p. 145

[138] M. M. Lehman et al. (1997). “Metrics and Laws of Software Evolution – The Nineties View.” Proceedings of the International Software Metrics Symposium, pp. 20–32. IEEE. ISBN 0-8186-8093-8. doi:10.1109/METRIC.1997.637156 p. 145

[139] Y. S. Lincoln, E. G. Guba (1985). *Naturalistic Inquiry*. Sage. ISBN 0-8039-2431-3. p. 111

[140] A. Lisitsa, I. Potapov (2005). “Temporal Logic with Predicate λ-Abstraction.” Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME), pp. 147–155. IEEE. ISBN 0-7695-2370-6. doi:10.1109/TIME.2005.34 p. 29

[141] M. Lombard, J. Snyder-Duch, C. C. Bracken (2002). “Content Analysis in Mass Communication: Assessment and Reporting of Intercoder Reliability.” *Human Communication Research* 28(4): 587–604. doi:10.1111/j.1468-2958.2002.tb00826.x p. 106

[142] M. Lombard, J. Snyder-Duch, C. C. Bracken (2003). “Correction.” *Human Communication Research* 29(3): 469–472. doi:10.1111/j.1468-2958.2003.tb00850.x p. 106

[143] L. M. MacLean, M. Meyer, A. Estable (2004). “Improving Accuracy of Transcripts in Qualitative Research.” *Qualitative Health Research* 14(1): 113–123. doi:10.1177/1049732303259804 p. 66

[144] J. Magee, N. Dulay, S. Eisenbach, J. Kramer (1995). “Specifying Distributed Software Architectures.” Proceedings of the European Software Engineering Conference (ESEC), LNCS, vol. 989, pp. 137–153. Springer. ISBN 3-540-60406-5. doi:10.1007/3-540-60406-5_12 p. 152

[145] S. T. March, G. F. Smith (1995). “Design and Natural Science Research on Information p. 7

Technology." *Decision Support Systems* 15(4): 251–266. doi:10.1016/0167-9236(94)00041-2

[146] F. Maris, P. Régnier (2008). "TLP-GP: Solving Temporally-Expressive Planning Problems." Proceedings of the International Symposium on Temporal Representation on Reasoning (TIME), pp. 137–144. IEEE. ISBN 978-0-7695-3181-6. doi:10.1109/TIME.2008.19 p. 141

[147] N. Markey, P. Schnoebelen (2003). "Model Checking a Path." Proceedings of the International Conference on Concurrency Theory (CONCUR), LNCS, vol. 2761, pp. 251–265. Springer. ISBN 3-540-40753-7. doi:10.1007/978-3-540-45187-7_17 pp. 30, 31, 32, 37, 40, 155

[148] P. Mayring (2000). "Qualitative Content Analysis." *Forum: Qualitative Social Research*, 1(2). http://nbn-resolving.de/urn:nbn:de:0114-fqs0002204 pp. 69, 73

[149] P. Mayring (2002). "Qualitative Content Analysis—Research Instrument or Mode of Interpretation?" In *The Role of the Researcher in Qualitative Psychology* (ed. M. Kiegelmann), pp. 140–149. Ingeborg Huber. ISBN 3-9806975-3-3. http://psydok.sulb.uni-saarland.de/volltexte/2007/943/ p. 73

[150] P. Mayring (2010). *Qualitative Inhaltsanalyse: Grundlagen und Techniken*. Beltz, 11th edn. ISBN 978-3-407-25533-4. p. 69

[151] D. McDermott et al. (1998). "PDDL—The Planning Domain Definition Language, Version 1.2." Tech. Rep. TR-98-003, Center for Computational Vision and Control, Yale University. p. 127

[152] E. McLellan-Lemal (2008). "Transcribing Data for Team-Based Research." In *Handbook for Team-Based Qualitative Research* (eds. G. Guest, K. M. MacQueen), pp. 101–118. AltaMira. ISBN 0-7591-0910-9. p. 66

[153] J. U. McNeal, M. F. Ji (2003). "Children's Visual Memory of Packaging." *Journal of Consumer Marketing* 20(5): 400–427. doi:10.1108/07363760310489652 p. 77

[154] N. Medvidovic, R. N. Taylor (2000). "A Classification and Comparison Framework for Software Architecture Description Languages." *IEEE Transactions on Software Engineering* 26(1): 70–93. doi:10.1109/32.825767 p. 12

[155] T. Mens (2010). "Model Transformation: A Survey of the State of the Art." In *Model-Driven Engineering for Distributed Real-Time Systems* (eds. J.-P. Babau et al.), pp. 1–19. ISTE. ISBN 978-1-84821-115-5. doi:10.1002/9781118558096.ch1 p. 169

[156] T. Mens, P. Van Gorp (2006). "A Taxonomy of Model Transformation." Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), ENTCS, vol. 152, pp. 125–142. Elsevier. doi:10.1016/j.entcs.2005.10.021 p. 169

[157] I. Mero-Jaffe (2011). "'Is That What I Said?' Interview Transcript Approval by Participants: An Aspect of Ethics in Qualitative Research." *International Journal of Qualitative Methods* 10(3): 231–247. http://ejournals.library.ualberta.ca/index.php/IJQM/article/view/8449 p. 66

[158] M. B. Miles, A. M. Huberman (1994). *Qualitative Data Analysis: An Expanded Sourcebook*. Sage, 2nd edn. ISBN 0-8039-5540-5. p. 111

[159] R. T. Monroe (2001). "Capturing Software Architecture Design Expertise with Armani, Version 2.3." Tech. Rep. CMU-CS-98-163R, Carnegie Mellon University, Pittsburgh. http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-163R.pdf p. 26

[160] F. Mosteller, D. L. Wallace (1963). "Inference in an Authorship Problem." *Journal of the American Statistical Association* 58(302): 275–309. doi:10.1080/01621459.1963.10500849 p. 67

[161] N. Müller, J. S. Damico (2002). "A Transcription Toolkit: Theoretical and Clinical Considerations." *Clinical Linguistics & Phonetics* 16(5): 299–316. doi:10.1080/02699200210135901 p. 66

[162] G. C. Murphy, D. Notkin, K. J. Sullivan (2001). "Software Reflexion Models: Bridging the Gap between Design and Implementation." *IEEE Transactions on Software Engineering* 27(4): 364–380. doi:10.1109/32.917525 p. 159

[163] L. Needels (2006). "DSMS Information Systems Architecture (DISA) Overview." Cleared Document 06-0975, Jet Propulsion Laboratory, Pasadena, CA. http://hdl.handle.net/2014/39174 p. 42

[164] K. A. Neuendorf (2002). *The Content Analysis Guidebook*. Sage. ISBN 0-7619-1978-3. ch. 5 passim

[165] M. Niazi, D. Wilson, D. Zowghi (2005). "A Maturity Model for the Implementation of Software Process Improvement: An Empirical Study." *Journal of Systems and Software* 74(2): 155–172. doi:10.1016/j.jss.2003.10.017 p. 67

[166] No Magic, Inc. (2011). *MagicDraw Open API Version 17.0.1 User Guide*. http://nomagic.com/files/manuals/MagicDraw%20OpenAPI%20UserGuide.pdf p. 121

[167] R. L. Nord, I. Ozkaya, P. Kruchten, M. Gonzalez-Rojas (2012). "In Search of a Metric for Managing Architectural Technical Debt." Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA), pp. 91–100. IEEE. ISBN 978-0-7695-4827-2. doi:10.1109/WICSA-ECSA.212.17 p. 170

[168] D. G. Oliver, J. M. Serovich, T. L. Mason (2005). "Constraints and Opportunities with Interview Transcription: Towards Reflection in Qualitative Research." *Social Forces* 84(2): 1273–1289. doi:10.1353/sof.2006.0023 p. 66

[169] OMG (2004). *Architecture-Driven Modernization (ADM)*. http://adm.omg.org/ p. 151

[170] OMG (2008). *Architecture Driven Modernization (ADM) Knowledge Discovery Meta-model (KDM)*. http://www.omg.org/spec/KDM/ p. 151

[171] OMG (2008). *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)*. http://www.omg.org/spec/QVT/ p. 169

[172] OMG (2010). *Object Constraint Language, Version 2.2*. http://www.omg.org/spec/OCL/2.2/ p. 50

[173] OMG (2010). *SysML 1.2*. http://www.omg.org/spec/SysML/1.2/ pp. 13, 47

[174] OMG (2010). *UML 2.3*. http://www.omg.org/spec/UML/2.3/ p. 13

[175] W. F. Opdyke, R. E. Johnson (1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems." Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA), pp. 145–160. Marist College. p. 145

[176] F. Oquendo et al. (2004). "ArchWare: Architecting Evolvable Software." Proceedings of the European Conference on Software Architecture (ECSA), LNCS, vol. 3047, pp. 257–271. Springer. ISBN 3-540-22000-3. doi:10.1007/978-3-540-24769-2_23 p. 147

[177] I. Ozkaya, R. Kazman, M. Klein (2007). "Quality-Attribute-Based Economic Valuation of Architectural Patterns." Tech. Rep. CMU/SEI-2007-TR-003, Software Engineering Institute, Pittsburgh. http://www.sei.cmu.edu/reports/07tr003.pdf p. 150

[178] I. Ozkaya, P. Wallin, J. Axelsson (2010). "Architecture Knowledge Management during System Evolution—Observations from Practitioners." Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge (SHARK), pp. 52–59. ACM. ISBN 978-1-60558-967-1. doi:10.1145/1833335.1833343 p. 91

[179] D. L. Parnas (1971). "Information Distribution Aspects of Design Methodology." Proceedings of IFIP Congress, pp. 339–344. North-Holland. ISBN 0-7204-2063-6. p. 145

[180] M. C. Paulk et al. (1993). "Key Practices of the Capability Maturity Model, Version 1.1." Tech. Rep. CMU/SEI-93-TR-025, Software Engineering Institute, Pittsburgh. http://www.sei.cmu.edu/reports/93tr025.pdf p. 146

[181] R. Pérez-Castillo, I. García-Rodriguez de Guzmán, M. Piattini, C. Ebert (2011). "Reengineering Technologies." *IEEE Software* 28(6): 13–17. doi:10.1109/MS.2011.145 p. 151

[182] D. E. Perry, A. L. Wolf (1992). "Foundations for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes* 17(4): 40–42. doi:10.1145/141874.141884 p. 12

[183] D. Phillips (1998). *The Software Project Manager's Handbook: Principles That Work at Work.* IEEE. ISBN 0-8186-8300-7. p. 146

[184] R. Popping (1988). "On Agreement Indices for Nominal Data." In *Sociometric Research* (eds. W. E. Saris, I. N. Gallhofer), vol. 1, pp. 90–105. St. Martin's Press. ISBN 0-312-00419-2. p. 102

[185] W. R. Powers (2005). *Transcription Techniques for the Spoken Word.* AltaMira. ISBN 0-7591-0843-9. p. 66

[186] L. H. Putnam (1978). "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." *IEEE Transactions on Software Engineering* SE-4: 345–361. doi:10.1109/TSE.1978.231521 p. 146

[187] J. Richardson (1992). "Supporting Lists in a Data Model (A Timely Approach)." Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 127–138. Morgan Kaufmann. ISBN 1-55860-151-1. http://www.vldb.org/conf/1992/P127.PDF p. 29

[188] D. Riffe, S. Lacy, F. G. Fico (1998). *Analyzing Media Messages: Using Quantitative Content Analysis in Research.* Lawrence Erlbaum Associates. ISBN 0-8058-2018-3. p. 107

[189] J. Ritsert (1972). *Inhaltsanalyse und Ideologiekritik: Ein Versuch über kritische Sozialforschung.* Athenäum. ISBN 3-7610-5801-2. p. 105

[190] E. M. Rogers (1997). *A History of Communication Study: A Biographical Approach.* Free Press, 1st paperback edn. ISBN 0-684-84001-4. p. 67

[191] N. Sadou, D. Tamzalit, M. Oussalah (2005). "A Unified Approach for Software Architecture Evolution at Different Abstraction Levels." Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), pp. 65–70. IEEE. doi:10.1109/IWPSE.2005.4 p. 149

[192] J. Saldaña (2013). *The Coding Manual for Qualitative Researchers.* Sage, 2nd edn. ISBN 978-1-4462-4737-2. p. 71

[193] M. Sandelowski (1994). "Notes on Transcription." *Research in Nursing & Health* 17(4): 311–314. doi:10.1002/nur.4770170410 p. 66

[194] A. Sanders (2010). "Innovation at JPL—GDS Modernization: A Case Study." Presented at the Ground Systems Architecture Workshop (GSAW). http://csse.usc.edu/GSAW/gsaw2010/s11b/sanders.pdf p. 42

[195] M. Schreier (2012). *Qualitative Content Analysis in Practice.* Sage. ISBN 978-1-84920-593-1. ch. 5 passim

[196] I. S. Schwartz, D. M. Baer (1991). "Social Validity Assessments: Is Current Practice State of the Art?" *Journal of Applied Behavior Analysis* 24(2): 189–204. doi:10.1901/jaba.1991.24-189 p. 107

[197] G. Sciavicco (2012). "Reasoning with Time Intervals: A Logical and Computational Perspective." *ISRN Artificial Intelligence.* doi:10.5402/2012/616087 p. 141

[198] W. A. Scott (1955). "Reliability of Content Analysis: The Case of Nominal Scale Coding." *Public Opinion Quarterly* 19(3): 321–325. doi:10.1086/266577 p. 102

[199] C. Seale (1999). *The Quality of Qualitative Research.* Sage. ISBN 0-7619-5597-6. pp. 110, 111

[200] M. Sefika, A. Sane, R. H. Campbell (1996). "Monitoring Compliance of a Software System with Its High-Level Design Models." Proceedings of the International Conference on Software Engineering (ICSE), pp. 387–396. IEEE. ISBN 0-8186-7247-1. doi:10.1109/ICSE.1996.493433 p. 159

[201] S. Sendall, V. Kozaczynski (2003). "Model Transformation: The Heart and Soul of Model-Driven Software Development." *IEEE Software* 20(5): 42–45. doi:10.1109/MS.2003.1231150 p. 169

[202] M. Shaw, D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall. ISBN 0-13-182957-2. pp. 12, 22

[203] Y. Shen, N. H. Madhavji (2006). "ESDM—A Method for Developing Evolutionary Scenarios for Analysing the Impact of Historical Changes on Architectural Elements." Proceedings of the International Conference on Software Maintenance (ICSM), pp. 45–54. IEEE. ISBN 0-7695-2354-4. doi:10.1109/ICSM.2006.26 p. 148

[204] M. Shepperd, C. Schofield (1997). "Estimating Software Project Effort Using Analogies." *IEEE Transactions on Software Engineering* 23: 736–743. doi:10.1109/32.637387 p. 146

[205] A. P. Sistla, E. M. Clarke (1985). "The Complexity of Propositional Linear Temporal Logics." *Journal of the ACM* 32(3): 733–749. doi:10.1145/3828.3837 p. 32

[206] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, J. E. Hannay (2008). "Building Theories in Software Engineering." In *Guide to Advanced Empirical Software Engineering* (eds. F. Shull, J. Singer, D. I. K. Sjøberg), pp. 312–336. Springer. ISBN 978-1-84800-043-8. doi:10.1007/978-1-84800-044-5_12 p. 7

[207] B. Spitznagel, D. Garlan (2001). "A Compositional Approach for Constructing Connectors." Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 148–157. IEEE. ISBN 0-7695-1360-3. doi:10.1109/WICSA.2001.948424 pp. 148, 149

[208] B. Spitznagel, D. Garlan (2003). "A Compositional Formalization of Connector Wrappers." Proceedings of the International Conference on Software Engineering (ICSE), pp. 374–384. IEEE. ISBN 0-7695-1877-X. doi:10.1109/ICSE.2003.1201216 pp. 148, 149

[209] I. Steinke (2004). "Quality Criteria in Qualitative Research." In *A Companion to Qualitative Research* (eds. U. Flick, E. von Kardorff, I. Steinke), pp. 184–190. Sage. ISBN 0-7619-7375-3. p. 105

[210] STORES Media (2013). "2012 Top 250 Global Retailers." http://www.stores.org/2012/Top-250-List p. 61

[211] STORES Media (2013). "2013 Top 100 Retailers." http://www.stores.org/2013/Top-100-Retailers?order=field_revenue_value&sort=desc p. 61

[212] K. F. Sturdevant (2007). "Cruisin' and Chillin': Testing the Java-Based Distributed Ground Data System 'Chill' with CruiseControl." Proceedings of the IEEE Aerospace Conference. IEEE. ISBN 1-4244-0525-4. doi:10.1109/AERO.2007.352957 p. 44

[213] E. B. Swanson (1976). "The Dimensions of Maintenance." Proceedings of the International Conference on Software Engineering (ICSE), pp. 492–497. IEEE. p. 90

[214] D. Tamzalit, T. Mens (2010). "Guiding Architectural Restructuring through Architectural Styles." Proceedings of the IEEE International Conference & Workshops on Engineering of Computer-Based Systems (ECBS 2010), pp. 69–78. IEEE. ISBN 978-0-7695-4005-4. doi:10.1109/ECBS.2010.15 pp. 149, 150

[215] D. Tamzalit, N. Sadou, M. Oussalah (2007). "Connectors Conveying Software Architecture Evolution." Proceedings of the International Computer Software & Applications Conference (COMPSAC), vol. I, pp. 391–396. IEEE. ISBN 0-7695-2870-8. doi:10.1109/COMPSAC.2007.97 p. 149

[216] P. Tarvainen (2007). "Adaptability Evaluation of Software Architectures; A Case Study." Proceedings of the International Computer Software & Applications Conference (COMPSAC), vol. II, pp. 579–586. IEEE. ISBN 0-7695-2870-8. doi:10.1109/COMPSAC.2007.240 p. 148

[217] R. N. Taylor et al. (1996). "A Component- and Message-Based Architectural Style for GUI Software." *IEEE Transactions on Software Engineering* 22: 390–406. doi:10.1109/32.508313 p. 152

[218] A. van Deursen, E. Visser, J. Warmer (2007). "Model-Driven Software Evolution: A Research Agenda." Proceedings of the Workshop on Model-Driven Software Evolution (MoDSE), pp. 41–49. http://www.cs.vu.nl/csmr2007/workshops/p19.pdf p. 151

[219] J. C. van Niekerk, J. D. Roode (2009). "Glaserian and Straussian Grounded Theory: Similar or Completely Different?" Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT), pp. 96–103. ACM. ISBN 978-1-60558-643-4. doi:10.1145/1632149.1632163 p. 74

[220] R. P. Weber (1985). *Basic Content Analysis.* No. 07-049 in Quantitative Applications in the Social Sciences, Sage. ISBN 0-8039-2448-8. p. 107

[221] M. Wermelinger, J. L. Fiadeiro (2002). "A Graph Transformation Approach to Software Architecture Reconfiguration." *Science of Computer Programming* 44: 133–155. doi:10.1016/S0167-6423(02)00036-9 pp. 148, 149

[222] B. J. Williams, J. C. Carver (2010). "Characterizing Software Architecture Changes: A Systematic Review." *Information and Software Technology* 52(1): 31–51. doi:10.1016/j.infsof.2009.07.002 p. 91

[223] R. Winter, R. Fischer (2006). "Essential Layers, Artifacts, and Dependencies of Enterprise Architecture." Proceedings of the IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW), paper 30. IEEE. ISBN 0-7695-2743-4. doi:10.1109/EDOCW.2006.33 p. 63

[224] M. M. Wolf (1978). "Social Validity: The Case for Subjective Measurement, or How Applied Behavior Analysis Is Finding Its Heart." *Journal of Applied Behavior Analysis* 11(2): 203–214. doi:10.1901/jaba.1978.11-203 p. 107

[225] C.-S. Wu, D. B. Simmons (2000). "Software Project Planning Associate (SPPA): A Knowledge-Based Approach for Dynamic Software Project Planning and Tracking." Proceedings of the International Computer Software & Applications Conference (COMPSAC), pp. 305–310. IEEE. ISBN 0-7695-0792-1. doi:10.1109/CMPSAC.2000.884739 p. 146

[226] R. K. Yin (2009). *Case Study Research: Design and Methods.* Sage, 4th edn. ISBN 978-1-4129-6099-1. pp. 59, 61, 111

[227] E. Yourdon, L. L. Constantine (1979). *Structured Design.* Prentice Hall. ISBN 0-13-854471-9. p. 145

[228] K. Yskout, R. Scandariato, W. Joosen (in press). "Change Patterns: Co-Evolving Requirements and Architecture." *Software and Systems Modeling.* doi:10.1007/s10270-012-0276-6 p. 150

[229] S. S. Yu et al. (2003). *Migrating to WebSphere V5.0.* Redbooks, IBM, 2nd edn. ISBN 0-7384-5338-2. p. 152

[230] A. Zalewski, S. Kijas, D. Sokołowska (2011). "Capturing Architecture Evolution with Maps of Architectural Decisions 2.0." Proceedings of the European Conference on Software Architecture (ECSA), LNCS, vol. 6903, pp. 83–96. Springer. ISBN 978-3-642-23797-3. doi:10.1007/978-3-642-23798-0_9 p. 151